ADDING VISUAL AND INTERACTIVE-SYNTAX TO TEXTUAL PROGRAMS

LEIF ANDERSEN

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Khoury College of Computer Sciences
Northeastern University
Boston, Massachusetts

August 2022
Leif Andersen:

Adding Visual and Interactive-Syntax to Textual Programs,
Doctor of Philosophy, Northeastern University, Boston, Massachusetts
© August 2022
Thesis Title: Adding Visual and Interactive Syntax to Textual Code

Author: Leif Andersen

PhD Program: ✔ Computer Science  ☐ Cybersecurity  ☐ Personal Health Informatics

PhD Thesis Approval to complete all degree requirements for the above PhD program.

Thesis Advisor

Jan Vitek

Robby Findler

KHOURY COLLEGE APPROVAL:

Associate Dean for Graduate Programs

Distribution: Once completed, this form should be scanned and attached to the front of the electronic dissertation document (page 1). An electronic version of the document can then be uploaded to the Northeastern University-UMI Website.
ABSTRACT

Many programming problems call for turning geometric thoughts into code: tables, hierarchical structures, nests of objects, trees, graphs, etc. Linear text does not do justice to such thoughts. But, it has been the dominant programming medium for the past and will remain so for the foreseeable future. While visual languages are a better medium for these thoughts, they lack the flexibility offered by linear text.

Hybrid visual-textual languages offer the best of both worlds. Programs written in a hybrid language can employ visuals when appropriate, while retaining the flexibility of text. Previous attempts at creating hybrid media have all been extra-linguistic; instead of supporting visual-interactive elements as language constructs, these media tied programming to one specific IDE. The biggest downside of such approaches is that programmers are unable to edit the textual portion of their programs using their preferred text editor.

This dissertation presents VISr (Visual and Interactive Syntax realized), a technique for adding a mechanism to existing programming languages that empowers programmers to extend it with domain-specific, visual and interactive elements. It presents two such realizations: one for Racket and one for ClojureScript. The dissertation also introduces two IDEs that can render interactive syntax elements as graphical-user interfaces. Specifically, it explains how to adapt Dr-Racket to visual-interactive syntax; and it introduces a new, browser-based IDE, specifically created for hybrid visual-textual programming: elIDE.

In support of the design, this dissertation also presents evidence of the usefulness and usability of VISr. The evidence comes from a user-facing evaluation and several case-studies of programs created using interactive-syntax extensions, including one extended case study using interactive-syntax extensions for video production.
First and foremost, I would like to thank my wife, Elizabeth Grimm. You are always there for me, you help me stay organized, and you always manage to put a smile on my face. Thank you Elizabeth, I love you more than words can say. I’d also like to thank the rest of my family for their support, and especially Edward Grimm, who does so much to help around the house. In addition, I would like to give a shout out to our cats, Charlie and Emma, who manage to keep life furry.

I would also like to thank my advisor, Matthias Felleisen. You have given me so many opportunities. Thank you for helping me find the words for the ideas I want to express; it means so much to me.

Next, I would like to thank the rest of my committee members: Arjun Guha, Jan Vitek, and Robby Findler. Your feedback has improved both this dissertation and the direction of my research.

While I am at it, I would like to thank my recent coauthors and other collaborators: Stephen Chang, Michael Ballantyne, and Cameron Moy. The long discussions I’ve had with you helped me find ideas I would not have found on my own.

I would also like to thank the current and past members of the Northeastern Programming Research Laboratory. They have provided a great space to work in these past few years. Especially Benjamin Greenman, Benjamin Chung, Ming-Ho Yee, Asumu Takikawa, and Vincent St-Amour. You have done so much to provide support and help me develop professionally.

On that note, I would like to thank both the Racket and Clojure communities. You have both provided much needed technical information that made all of this work possible. On top of that, your encouragement has helped keep me going.

Additionally, I would like to acknowledge the support I got during my undergraduate experience back in Utah. Thank you to the University of Utah’s School of Computing, and College of Engineering Faculty and Staff. You have all done so much to teach me about computer science. I would not be where I am today without your support. Also thank you to Matt Might, Andy Keep, and William Byrd, for introducing me to the wonderful world of Programming Languages.

Finally, I would like to especially thank Adair Kovac and Bruce Stagbrook de Clairmont. You two took care of me at some of my lowest points and helped me find myself. You will always be dear to me for that.
For most of my years at Northeastern, I have been supported on grants by the National Science Foundation (in particular SHF 1421412, CRI 1823244, and SHF 2007686).
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Thesis</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Outline</td>
<td>2</td>
</tr>
<tr>
<td>I Introducing Visual and Interactive-Syntax</td>
<td></td>
</tr>
<tr>
<td>2 Tsuro: A Board Game Example</td>
<td>7</td>
</tr>
<tr>
<td>3 Design Goals</td>
<td></td>
</tr>
<tr>
<td>3.1 Design Requirements</td>
<td>11</td>
</tr>
<tr>
<td>3.2 Design Desiderata</td>
<td>12</td>
</tr>
<tr>
<td>3.3 Programming Workflows</td>
<td>13</td>
</tr>
<tr>
<td>4 Constructing Interactive Syntax</td>
<td></td>
</tr>
<tr>
<td>4.1 Some Basic Background on Syntax Extensions</td>
<td>17</td>
</tr>
<tr>
<td>4.2 Editing As a Phase</td>
<td>18</td>
</tr>
<tr>
<td>4.3 Bridging the Gap Between Edit Time, Compile Time, and Run Time</td>
<td>19</td>
</tr>
<tr>
<td>4.4 Edit-Time Programming, a Tsuro Example</td>
<td>21</td>
</tr>
<tr>
<td>5 A Plethora of Examples</td>
<td></td>
</tr>
<tr>
<td>5.1 Examples of Diagram Documentation</td>
<td>25</td>
</tr>
<tr>
<td>5.2 The Expressive Power of interactive syntax</td>
<td>26</td>
</tr>
<tr>
<td>5.3 In Depth: Red-Black Trees</td>
<td>28</td>
</tr>
<tr>
<td>5.4 In Depth: Form Builders</td>
<td>29</td>
</tr>
<tr>
<td>6 The Implementation of racket/visr</td>
<td></td>
</tr>
<tr>
<td>6.1 Ingredients for an Interactive-Syntax System</td>
<td>33</td>
</tr>
<tr>
<td>The Editor Form</td>
<td>33</td>
</tr>
<tr>
<td>Edit-time semantics</td>
<td>34</td>
</tr>
<tr>
<td>Editor Elaboration</td>
<td>35</td>
</tr>
<tr>
<td>Cooperating With an IDE</td>
<td>35</td>
</tr>
<tr>
<td>6.2 A prototype using Racket and DrRacket</td>
<td>36</td>
</tr>
<tr>
<td>The Editor Form</td>
<td>36</td>
</tr>
<tr>
<td>Edit-time semantics</td>
<td>36</td>
</tr>
<tr>
<td>Editor Elaboration</td>
<td>38</td>
</tr>
<tr>
<td>Cooperating With an IDE</td>
<td>38</td>
</tr>
<tr>
<td>7 Shortcomings with the Prototype</td>
<td>41</td>
</tr>
<tr>
<td>II A Usable Implementation for Interactive-Syntax</td>
<td></td>
</tr>
<tr>
<td>8 Problems of Usability</td>
<td>45</td>
</tr>
<tr>
<td>9 Design</td>
<td></td>
</tr>
<tr>
<td>9.1 Architecture overview</td>
<td>47</td>
</tr>
<tr>
<td>9.2 Evaluating ClojureScript in a Browser</td>
<td>48</td>
</tr>
<tr>
<td>9.3 The elIDE IDE</td>
<td>49</td>
</tr>
<tr>
<td>10 Tsuro Reloaded</td>
<td></td>
</tr>
<tr>
<td>10.1 The Tsuro Editor</td>
<td>51</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>22.1</td>
<td>Creating Languages, the Racket Way</td>
</tr>
<tr>
<td>22.2</td>
<td>The Essence of Video’s Syntax</td>
</tr>
<tr>
<td>22.3</td>
<td>Video, Behind the Scenes</td>
</tr>
<tr>
<td>22.4</td>
<td>The Preview Editor</td>
</tr>
</tbody>
</table>

V Final Remarks

23 Related Works | 123
23.1 Edit Time | 123
23.2 Graphical and Live Languages | 124
23.3 Bidirectional and Projectional Editing | 125
23.4 Sandblocks | 126
23.5 Video Production | 129

24 Future Work | 131
24.1 Edit-Time Code and Phasing | 131
24.2 Usability, Usefulness, and Practicality | 131

25 Conclusion | 133

Bibliography | 135
INTRODUCTION

The ultimate goal of any programming language is to allow programmers to clearly communicate their thoughts to other programmers, as well as machines. Text, a common medium used to communicate thoughts for several millennia, works as a natural medium for writing programs in most situations; for most of its history, programming has been limited to text. People know, however, that the occasional picture “is worth a thousand words” [Flanders 1911].

A purely textual medium does limit a language’s ability to express geometric concepts, while a purely graphical medium limits a language’s ability to express concepts best described with text. Although pure visual programming languages do occasionally emerge, they are usually limited to a fixed domain: pedagogy, UI design, algebraic systems, etc. General purpose visual languages do not exist.

A hybrid of textual and visual programming can enable programming languages to support visual concepts, while keeping the expressivity and effectiveness of a textual language. For example, when implementing a network protocol backed by a state machine, this hybrid approach enables a graphical presentation of the state machine mixed with a textual presentation of the protocol’s effects. Using a purely textual representation gives no insight into the state machine; likewise using a purely visual representation conflates the state machine with the remaining program logic.

Previous attempts to realize a hybrid visual-textual language (see chapter 23) have had little success. While promising, each attempt had a number of flaws limiting adoption. In particular, previous attempts were either not extensible, that is, limited to a fixed set of visualized constructs; or they abandoned plain text as a storage medium, meaning all development was limited to a custom development environment. Fortunately, neither of these drawbacks are necessarily fundamental to the concept of hybrid visual-textual languages.

1.1 THESIS

It is possible to construct a usable and useful mechanism for programmatically adding visual and interactive, domain-specific syntax to an existing textual programming language.

This dissertation presents several designs and implementations in support of my thesis. A hybrid approach enables mixing graphical and textual pieces of code. It empowers programmers to add graphical domain-specific syntax for their particular problems as needed, which is
key to making such languages useful. Then programmers can embed these graphical syntax components in textual ones and visa versa. This design also allows programmers to create new notations directly within their programs without requiring an external development environment.

In the context of a PL system, usability [Lazar et al. 2017] refers to the question of whether developers can actually use the system to write programs. If programmers can use these languages to write a small game, for example, then they are usable. Usability is not concerned with creating an optimal development-time experience or producing the most efficient run-time code. Rather, usability is concerned with creating a system that people can use to produce and run programs effectively. Some case studies will suffice to demonstrate usability.

1.2 OUTLINE

My dissertation research supports my thesis with both partial failures and successes. The rest of the dissertation reports on the research in four parts.

The first part is a description of visual-interactive syntax, its components, and its usefulness. Continuing on, the second part demonstrates the usability of visual-interactive syntax in the context of browser and DOM environments. Next, the third part expands this usability to languages that are not browser based. The fourth part is an extended case study using interactive syntax in combination with traditional language extension for video production. Finally, this dissertation concludes with related and future work.

More specifically, the first part is organized as follows:

- Chapter 2 gives a concrete example of interactive syntax and its benefits compared to textual syntax.

- Chapter 3 takes an abstract look at the design space of source code manipulation in software engineering and explains how interactive syntax must fit into it.

- Building on the previous two chapters, chapter 4 outlines the components of interactive-syntax extensions and shows how to insert instances into code.

- Chapter 5 takes this further with several interactive-syntax related case studies. This chapter also reports a brief evaluation of the communicative properties of interactive-syntax.

- Using this design, chapter 6 sketches how an existing programming language can be enriched with interactive-syntax capabilities using the design of chapter 4.
Chapter 7 outlines the serious shortcomings of the implementation described in the first part.

Moving onto the second part, the dissertation presents a usable design:

• Chapter 8 takes a step back by describing usability in the context of software development and the requirements for interactive syntax to be usable.

• Using this experience, chapter 9 presents an alternate design for visual-interactive syntax that is usable.

• Chapter 11 sketches some implementation details for that design.

• To prove this design’s usability, chapter 12 presents the results from a user facing evaluation.

• Chapter 13 supplies additional evidence in the form of several small case studies.

• Finally, chapter 14 briefly compares this second prototype with the one from the first part.

Taking the usefulness of the first part and the usability of the second part, the third part generalizes this design:

• Chapter 15 broadens the scope of the usable architecture to languages without a browser component.

• With this broader scope, chapter 16 describes an architecture that allows interactive-syntax extensions to be used across all languages that can access a DOM-based GUI framework.

• Chapter 17 provides a case study for using this architecture to remake the example from chapter 2.

• To show this architecture is still usable, chapter 18 presents a case study where another programmer used this cross-language approach to make a visual semantics for the Simply Typed Lambda Calculus.

The fourth part is an extended case study:

• Chapter 19 makes the case for using programming languages with visual and interactive syntax as a tool for producing video.

• Using this motivation, chapter 20 describes a small language that meets these requirements.

• Chapter 21 ties back to interactive-syntax by demonstrating how this visual-interactive domain-specific language comes with the benefits of using both a graphical video editor as well as a programming language.
• Finally, chapter 22 shows how to implement this design.

Wrapping things up, the fifth part presents related and future work:

• Chapter 23 explains how my research compares to related work.
• This dissertation opens up several possible directions for future work, discussed in chapter 24.
• Chapter 25 gives a brief conclusion.
Part I

INTRODUCING VISUAL AND INTERACTIVE-SYNTAX

This part of the dissertation is derived from collaborative work with Ballantyne and Felleisen [Andersen et al. 2020]
Consider a software system that implements a game such as Tsuro.\(^1\) In this game, players take turns growing a graph from square tiles, each of which displays four path segments. A player places one avatar at an entry point on the periphery of the grid-shaped board. New tiles are added next to a player’s avatar, and all avatars bordering this new tile are moved as far as possible along the newly extended paths until they face an empty place again. If an avatar exits the board, its corresponding player is eliminated. The last surviving player wins.

Now imagine a programmer wishing to articulate unit tests in the context of a Tsuro implementation. When a mechanism for creating interactive and visual syntax is available, the tester may add a Tsuro tile as a new language construct. This developer creates an instance of this syntax via UI actions, i.e., key strokes or menu selection, and inserts it into the drawing context of the integrated development environment (IDE). A visual-syntax construct is referred to as an editor. Consider the image below, which shows a tile editor. It displays a graphical representation of the tile (left) and manual text entry fields (right). These text fields and graphical representation are linked [Becker et al. 1987; Wills 2008]. The graphic updates whenever a user updates the text; the text fields update when the programmer connects two nodes graphically via user interface (UI) actions. What is shown here is the state of this syntax just as the developer is about to connect the nodes labeled "C" and "D":

![Tile Editor](image)

A tile editor compiles to code that evaluates to a hash table representation of its bidirectional connections. For the above example, the hash table connects the "A" node with "G", as the following lookup operations confirm:

\(^1\) [https://en.wikipedia.org/wiki/Tsuro](https://en.wikipedia.org/wiki/Tsuro)
Note how the editor itself assumes the role of a hash table here.

A Tsuro developer will also create interactive-syntax for Tsuro boards, shown below. The board syntax supplies a grid of slots. Each slot is initially empty, but the programmer may place Tsuro tiles there to mimic players’ actions. Take a look at this example:

In this image, three players have already placed one tile each (the bottom row extreme left and extreme right plus the third slot in the top row), and each tile is occupied by an avatar. As far as run time is concerned, the Tsuro board editor evaluates to an object that contains a matrix of tiles.

Once a programmer has extended the language with these two Tsuro-specific language constructs, a unit test using these graphical editors looks as follows in code:

This unit test checks whether the addTile method works properly. The method expects an initial board state, a tile, and a player. Its result is a new board state with the tile placed in the slot that the player’s avatar faces in the given board state and with the avatar moved as far as possible so that it again faces an empty spot on the grid.

For comparison, figure 1 articulates the same unit test with plain textual code. As with the graphical unit test, addTile expects a board, tile, and player. The board is constructed with tiles and player start locations. Each tile is a list of eight letters, each representing its connecting node. I invite the readers to improve on this notational choice and compare their improvements with the visual syntax above.
Interactive visual syntax is just syntax, and syntax composes according to grammatical rules. An editor may appear within textual syntax, as shown above. And textual syntax may appear within interactive-syntax. Let's return to our Tsuro developer who may wish to write helper functions for unit tests that produce lists of board configurations for exploring moves. Here is such a function, again extracted from my code base:

```
; Tile -> [Listof Board]
(define (all-possible-configurations t)
  (for/list ([d DEGREES])
    ... (send t rotate d) ...))
```

As the type signature says, this function consumes a tile and generates a list of boards. Specifically, it (for) loops over a list of DEGREES, with each iteration generating an element of the resulting list (hence for/list). Each iteration generates a board by rotating the given tile \( t \) by \( d \) degrees and placing it in a fixed board context. The dots surrounding the method call are supposed to suggest this fixed context.

Once again, the developer can either express this context as a map like that of figure 1 or use an instance of interactive visual syntax. Figure 2 shows the second scenario. The spot on the board where the tile is to be inserted is a piece of interactive-syntax for editing code. The zoomed image on the right indicates how a developer manipulates this code. Clicking on this tile pops up a separate text editor. The developer manipulates code in this editor and closes the editor when the code is completed. Creating such an interactive Tsuro board is only
slightly more work than creating the one used for the unit test above. Ideally, in a usable programming language with interactive-syntax capabilities, a developer is able to re-use application graphical user interface (GUI) code inside of editors.

In sum, this example illustrates how the message sent to future maintainers is significantly clearer when visual syntax is added to plain text. Programmers are able to quickly get an image of the tile connections and see how the tiles jointly form paths. In contrast, any plain text variant requires developers to manually piece together the segments to reconstruct the intended layout. Furthermore, no information is lost by using the visual presentation. Developers still have access to textual mappings of each tile, as well as the layout of all tiles on the board. In essence, when done correctly, interactive syntax only adds clarity. The next chapter discusses the general design requirements for an interactive-syntax system implementation.
DESIGN GOALS

My research starts from an acknowledgment of the dominance of linear text. Its goal is to enable developers to supplement linear text with visual and interactive syntax as soon as they are tempted to document any code with some form of diagram or other non-textual illustration. Furthermore, the transition from linear text to visual and interactive syntax must be as smooth as possible. This chapter lays out three fundamental design requirements for such a programming language. Using these design requirements, it then lists specific design desiderata for a usable interactive-syntax language. Finally, this chapter enumerates all of the ways programmers work with code and how the design requirements for interactive syntax can accommodate these workflows.

3.1 DESIGN REQUIREMENTS

In order to gain widespread adoption, interactive syntax must satisfy three primary criteria. First, it is imperative to demonstrate the feasibility of mixing textual and interactive syntax in the context of an ordinary and existing programming language. Ordinary here refers to the kind of languages developers already use: with classes, objects, functions, and side effects. Likewise, existing means in use by a non-trivial community. Adding a visual-syntax mechanism to such a language is the simplest ways to tempt programmers into its use and to convince them of its usability as well as its usefulness.

Second, more than one specific IDE must accommodate interactive syntax. Indeed, programs in the revised language must not even preclude editing in purely text-oriented tools such as Vim. Developers frequently have strong preferences concerning the IDE they use. They are much less likely to allow teammates to use interactive syntax if it means everyone on the team must migrate from their favorite IDEs to the single, fixed IDE that supports interactive syntax.

Finally, developers must be able to amortize the investment in graphical user interfaces. The construction of interactive-syntax extensions demands code that implements simple GUIs, a potentially time-consuming task compared to, say, drawing ASCII diagrams. If these GUIs can share code with the actual graphical interface of the

---

1 The popularity of notebooks, REPLs, interactive prompts, and virtual machines demonstrate the usefulness of visual organizations for experimentation and data analysis. But, they also show how quickly such an approach becomes unwieldy as software systems grow [Chattopadhyay et al. 2020]. Techniques such as snapshots and cafés [Dybvig 2006] reduce the unwieldiness, but do not eliminate it.
software application, though, the cost of creating interactive-syntax extensions may look quite reasonable. The implication is that an interactive-syntax extension mechanism must use the existing GUI libraries of the chosen language as much as possible.

3.2 Design Desiderata

With these basic requirements in mind, I can now state specific design desiderata for interactive-syntax extensions themselves:

1. An interactive visual syntax is just syntax. It merely articulates an idea better than textual syntax. If the underlying grammar distinguishes between definitions, expressions, patterns, and other syntactic categories, it should be possible to use visual syntax in each such category.

2. Interactive syntax is persistent. The point of interactive syntax is that it permits developers to send a visual message across time. In contrast to wizards and code generators, it is not a GUI that pops up so that a developer can create textual code. Hence an editor must continuously serialize and save its state. One developer can then quit the IDE, and another can open this same file later, at which point the interactive-syntax editor can render itself after deserializing the saved state.

3. Interactive visual syntax constructs must compose with textual syntax according to the grammatical productions. As already demonstrated, this implies that textual syntax may contain interactive visual syntax and vice versa. In principle, developers should be able to nest visual and textual syntax arbitrarily deep.

4. The ideal mechanism implements a low-friction model for the definition and use of interactive visual syntax. A developer should be able to define and use an interactive-syntax extension in the same file. Indeed, this principle can be further extended to lexical scope. As with traditional syntax extension mechanisms, a developer should be able to define and use an interactive-syntax extension within a function, method, class, module, or any other form of code block that sets up a lexical scope.

5. The creator of interactive visual syntax must be able to exploit the entire language, including the extension mechanism itself. If the underlying language permits abstraction over syntax, like Rust, then a developer must be able to abstract over definitions of interactive visual syntax; in the same vein, an instance of interactive visual syntax may create new forms of visual and textual syntax abstractions.
6. Interactive visual syntax demands sandboxing for the IDE. The instantiation of interactive visual syntax into an editor runs code. When developers manipulate editors, code is run again. In an ordinary language, such code may have side effects. Hence, the extension mechanism must ensure that the code does not adversely affect the functioning of the IDE.

7. Interactive visual syntax demands sandboxing for code composition. Experience with syntax extension mechanisms [Flatt 2002] suggests that it is also desirable to isolate the execution of the edit-time code from other phases, such as the compilation phase and the runtime phase. This form of sandboxing greatly facilitates co-mingling static code from different phases without suffering from accidental dynamic interference.

These design desiderata describe goals for a language supporting interactive-syntax extensions to be useful and usable. A language that supports these goals allows developers to use interactive-syntax extensions in existing programs as well as create new types of extensions when needed. Likewise, these extensions will not interfere when developers are not using them or are even using an environment with no support for interactive-syntax extensions.

3.3 Programming Workflows

Programmers interact with their codebases in many different ways. The following list enumerates the types of interactions and explains whether and how the addition of interactive syntax affects these parts of the development workflow, Chapter 9 discusses how one implementation, core.visr, realizes each of these interactions:

- **Auditing** is the task of reading code. The direct goal of interactive syntax is to make this task easier.

- **Creation** is writing new code. Interactive-syntax extensions can be used for creating new code, likewise new code can be used to create interactive-syntax extensions.

- **Move and Copy** is both the act of copying to, and pasting from the clipboard, as well as a more direct drag and drop. Interactive syntax must also support duplicating and moving extensions.

- **Running** a program is another fundamental part of software development. Interactive-syntax extensions must not inhibit this.

- **Search and Replace** also referred to as Ctr + F is the act of finding a piece of code and optionally replacing it with new text. Traditionally visual syntax has failed here. By being represented as text, however, interactive syntax might be usable with traditional search tools.
For completeness, this list includes the remaining ways programmers interact with code:

- **Abstraction** requires a bit of code creation, comprehension, and moving. It is the task of generalizing a block of code to work in multiple situations. To work with abstraction, interactive-syntext must facilitate converting one type of interactive-syntext extension to another.

- **Autocomplete** requires semantic knowledge of the programming language. An extensive interactive-syntext language should allow extensions to support it.

- **Coaching** runs some analysis on the program and displays the results inline in the editor. This includes everything from underlining unbound variables to highlighting register spilling. A coaching system must know about a language’s semantics and thus can also directly support interactive syntax.

- **Code Folding** enables IDEs to hide irrelevant blocks of code while editing. Because interactive syntax can be placed in text, an IDE can support code-folding equally well for textual and visual syntax.

- **Comments** are part of a codebase that is not meant to be run. When possible, interactive syntax should not touch comments. Additionally, programmers should be able to add comments to interactive-syntext extensions.

- **Comparison** is often referred to as line-by-line diffing. It is the task of taking two blocks of code and creating a report of their difference. To support diffing, interactive syntax must either provide a tool to interpret interactive-syntext extensions as atoms/composites or serialize extensions into human readable text.

- **Debugging** involves running a program as individual steps that a human can move through sequentially; the human can inspect the state of the program during each step. While associated with editors, debuggers rely on understanding a language’s semantics. Therefore, in principle, interactive syntax naturally fits debuggers. Though more research is needed to prove this.

- **Dependency/Environment Update** is similar to migration, but requires no manual action from the programmer. Any changes made to an interactive-syntext extension’s definition is reflected in uses of that extension.

- **Elimination** is the dual of abstraction, and often requires inlining existing code. Interactive-syntext extensions must also be removable from source.
• Go to Definition/Use allows programmers to jump between definitions and uses in their programs. Again, IDEs must understand program semantics to handle this properly and should be possible using interactive-syntax extensions.

• Merging two blocks of code is the natural extension to comparison. Instead of viewing a report of the difference, however, merging attempts to generate running code. Likewise, programmers must be able to merge interactive-syntax extensions, both of the same type and of different types.

• Migration happens when a dependency or platform changes, breaking backwards compatibility. Frequently this requires small tweaks through an entire codebase. Supporting extension migration suffices here.

• Multi-Cursor Editing is when two or more developers concurrently edit the same piece of code in the same buffer. This is orthogonal to interactive syntax.

• Refactoring is a syntax aware search and replace. A refactoring tool can change the name of a lexical variable without changing other variables that happen to share the same name. Interactive syntax brings programming semantics to the task of editing. Thus it can support traditional refactoring.

• Reflow automatically transforms program text to conform to some style standards. This requires understanding a language’s semantics to do well. For interactive syntax, reflow is similar to code moving.

• Style changes how an IDE displays text without changing the text itself. This includes increasing or decreasing font size, or changing the color theme. While not strictly needed, an interactive-syntax editor can benefit from coordinating with an IDE theme.

• Undo/Redo is straightforward for text. For interactive syntax, each extension can package multiple changes into a single undo/redo step.

While each of these operations is important, the two most fundamental ones are auditing and creation. An environment that supports interactive-syntax extensions is usable if it enables programmers to do these two tasks. In the context of language extensions, creation can be subdivided into the creation of the language extension and uses of that extension.
CONSTRUCTING INTERACTIVE SYNTAX

Writing interactive-syntax extensions parallels the process of writing traditional syntax extensions. Both traditional and interactive-syntax extensions allow compile-time code and run-time code to co-exist in the same program, the same file, and even the same expression. Interactive syntax adds the additional notion of an edit-time phase, code that runs while the programmer edits the code.

This chapter describes interactive-syntax extensions as realized in Racket, its parallels to traditional syntax extensions [Felleisen et al. 2018], and how these extensions can implement the Tsuro tiles from chapter 2. The key novelty is define-interactive-syntax, a construct for creating new interactive-syntax forms. I call this implementation racket/visr.

4.1 SOME BASIC BACKGROUND ON SYNTAX EXTENSIONS

Racket comes with a highly expressive sub-language of macros that enable programmers to extend the language syntactically. To process a program, Racket’s reader creates a syntax tree, stored as a compile-time object. Next the macro expander traverses this tree and discovers Racket’s core forms while rewriting instances of macros into new syntax sub-trees. In order to realize this rewriting process, the expander partially expands all syntax trees in a module and then adds macro definitions to a table of macro rewriting rules for the second, full expansion pass.

Macros are functions from syntax trees to syntax trees. Instead of

\[
\text{define (f x) _ _ _ elided _ _ _}
\]

a programmer writes

\[
\text{define-syntax (m x) _ _ _ elided _ _ _}
\]

to define the syntax transformer \(m\). When the expander encounters the shape \((m _ _ _ elided _ _ _)\), it applies the transformer to this entire syntax tree. The transformer is expected to return a new syntax tree, and once this happens, the expander starts over with the traversal for this new one. While macros are often used to produce expressions and definitions, \((m _ _ _ elided _ _ _)\) may also expand to define-syntax and thus introduce new syntax definitions. These “on-the-fly” definitions are why the macro expander takes the one-and-a-half pass approach, described above, to elaborating modules into the Racket core language.
A programmer may specify a macro either as a declarative rewriting rule or as a procedural process. For the second variant, the macro may wish to rely on functions that are available at compile time. In Racket a module may import ordinary libraries `for-syntax`, or it may locally define functions to be available at compile time with `begin-for-syntax`. Thus,

```
(for-syntax (define (g a b c) _ _ _ elided _ _ _))
```

makes the ternary function `g` available to procedural macros.

Racketeers speak of the compile-time phase and the run-time phase. Here a **phase** is a syntactic separation that determines when code runs and provides a semantic separation of its effects [Flatt 2002]. Naturally, function definitions for the compile-time phase may call macros defined for the compile-time phase, which are in turn defined in the compile-time phase’s compile-time phase. Programmatically a module may thus look as follows:

```
#lang racket
(define-syntax (m x) _ _ _ (f a b) _ _ _)
(begin-for-syntax
(define (f y z) _ _ _ (k c d e) _ _ _)
(define-syntax (k w) _ _ _ (g) _ _ _)
(begin-for-syntax
(define (g) _ _ _ elided _ _ _)))
```

Phases in Racket programs are nested arbitrarily deep, because programmers appreciate this power.

## 4.2 Editing as a Phase

Our technical idea is to fold the **editing phase** into the programming language, specifically into the existing phase system of Racket. This extension to Racket consists of two linguistic constructs, analogous to `define-syntax` and `begin-for-syntax`:

- **define-interactive-syntax** creates and names an interactive-syntax extension. Roughly speaking, an interactive-syntax extension is a specialized graphical user interface defined as a class-like entity. It comes with one method of rendering itself in a graphical context, such as an IDE; a second for reacting to events (keyboard, mouse, etc.); a third for tracking local state; and a final one for turning the state into running code.

- **begin-for-interactive-syntax** specifies code that exists for the editing phase in an interactive-syntax extension. It can only appear at the top level, module level, inside other uses of the
begin-for-interactive-syntax form, and other places that a begin-for-syntax block is valid.

That is, an interactive-syntax extension executes during edit time yet denotes compile-time and run-time code.

The code in a begin-for-interactive-syntax block runs when the development environment opens the module for editing, as well as any time its content is modified. Like its cousin begin-for-syntax, the begin-for-interactive-syntax form is mostly used as a mechanism for definitions that are needed to define interactive-syntax.

In Tsuro, for example, a trace-player function calculates the path of a player’s token from the start position to the end position. The Tsuro-board editors use it to calculate where they should draw the tokens after the placement of a tile, meaning the function is needed at edit-time, too:

```racket
(begin-for-interactive-syntax
  (require "Model/player.rkt"))
```

Requiring an ordinary module at edit time imports its definitions into the desired scope at edit time.

4.3 Bridging the gap between edit time, compile time, and run time

The define-interactive-syntax form bridges the gap between run-time and edit-time code; i.e., it is analogous to define-syntax. In contrast to define-syntax, which allows run-time code and compile-time code to interact, the latter referencing and generating the former, the define-interactive-syntax form connects edit-time code with run-time code, generating the latter from the former.

A Racket syntax extension consists of a new grammar production and a translation into existing syntax. By contrast, interactive-syntax extensions consist of four different pieces:

1. a presentation, meaning a method for rendering its current state, runs at edit-time;

2. an interaction, that is, a method for reacting to direct manipulation actions, runs at edit-time;
3. a semantics, which is compile-time code that generates run-time code; and

4. persistent storage, i.e., a specification of persisted data and its external representation, which exists at both edit time and compile time.

Once an interactive-syntax definition exists, a developer may insert an instance into an IDE buffer by a UI action, such as a mouse click or button press. DrRacket [Findler and PLT 2010], an IDE bundled with the Racket distribution, implicitly places this editor within a begin-for-interactive-syntax block so that its edit-time code runs continuously during program development. When the programmer requests the execution of the module, the extension’s semantics turns the current state into run-time code, properly integrated into the lexical scope of its location. Opening the file in any other editor that does not support racket/visr results in a human readable textual representation of the editor.

Concretely the define-interactive-syntax form consists of four sub-forms, stored using Racket’s class system:

```
(define-interactive-syntax name$ base$
  (define/public (draw ctx)
    _ _ _ visually rendering _ _ _ )
  (define/public (on-event event)
    _ _ _ interactions code _ _ _ )
  (define-elaborator
    _ _ _ generating run-time code _ _ _ )
  ( _ _ _ persistent data _ _ _ ))
```

The first line specifies the name of the interactive syntax and from which base class it is derived. The ‘$’ in name$ and base$ is only a convention. The base$ class mimics its object-oriented counterpart, providing the interface for racket/visr to hook into. Like classes, interactive-syntax definitions benefit from implementation inheritance. The draw and on-event methods (lines 2–5) make up the forms’s user interface, code that heavily relies on Racket’s platform-independent graphical-user interfaces [Flatt et al. 2010] and tools for the interactive development environment [Cooper and Krishnamurthi 2004; Findler and PLT 2010]. For specifying the semantics of the new construct, a developer uses the meta-DSL for specifying text-based language extensions (lines 6 and 7). The remaining pieces (lines 8 and below) make up the persistent storage of the syntax form.

To support the specification and management of persistent state, my design also supplies a custom language extension. With this extension, a developer can articulate how to react to changes of the data, how to serialize it for future use, and how to deserialize data (if it exists) to resume the use of an interactive-syntax extension. The extension is
called \texttt{define-state} and its syntax is as follows:

\begin{verbatim}
(define-state name default
  state-properties ...)
\end{verbatim}

The properties describe these aspects of the state variables. First, they provide traditional getter/setter methods. Second, they provide an optional mechanism for users to marshal seemingly unserializable values into serializable ones. Finally, they describe how long a value must persist.

Consider the state for an extension that adds word processors to the language. That state might include the prose the user typed as well as the cursor’s position. The document’s getter/setter methods, as well as its serialization would be fairly pedestrian and the \texttt{define-state} form would handle this automatically. However, persistence is less trivial. The user may expect the text to be saved in the file, but not the current cursor position, but the user does expect the cursor to remain in place while the document is open.

Semantically, an interactive-syntax definition is a class that runs code both during edit time and compile time. Definitions using \texttt{define/public} are methods and capitalize on object inheritance. The \texttt{define-interactive-syntax} form ensures that \texttt{draw} and \texttt{on-event} are among the defined methods. The \texttt{define-state} form is mapped to the class fields. Finally, the \texttt{define-elaborator} form acts as a macro that generates run-time code. See chapter 6 for details.

4.4 \textsc{Edit-Time Programming, A Tsuro Example}

Implementing a dedicated state, renderer, event handler, and semantics for each interactive-syntax extensions is somewhat labor intensive. To reduce the work load, our prototype implementation supports the standard GUI creation techniques available in Racket. These techniques fit into three categories: inheritance, container editors, and graphical editors.

Developers use inheritance and mixins [Flatt et al. 2006] to write only absolutely necessary \texttt{draw} and \texttt{on-event} methods. Inheritance works just like in Java. For example, every editor extends a \texttt{base$} class, which supplies basic drawing and event handling. Mixin functions abstract over inheritance. The \texttt{define-interactive-syntax-mixin} form creates new mixin types. These mixins, like standard object-oriented mixins, are applied as functions to an interactive-syntax definition.

Container editors facilitate editor composition, which almost completely eliminates the need for manually creating methods for the resulting product. The three most predominant container blocks are \texttt{vertical-block$} for vertical alignment, \texttt{horizontal-block$} for horizontal alignment, and \texttt{pasteboard$} for free-flowing editors. Each of
(begin-for-interactive-syntax
  (define TILE-NODES (list "A" "B" "C" "D" "E" "F" "G" "H")))

(define-interactive-syntax tsuro-tile$ horizontal-block$ (super-new)
  ; STATE
  (define-state pairs (hash)
    #:elaborator #t
    #:getter #t)
  ; Char Char -> Void
  ; EFFECT connects letter to other and vice versa in pairs
  (define/public (connect! letter other)
    (send (hash-ref field-gui letter) set-text! other)
    (send (hash-ref field-gui other) set-text! letter)
    (set! pairs (hash-set* pairs letter other other letter))
    (send picture set-tile! (draw-tile pairs)))

  ; VIEW : two horizontally aligned elements
  (define picture (new tsuro-picture$ [parent this]
    [connections pairs]))

  (define fields (new vertical-block$ [parent this]))
  ; Char -> Void
  ; EFFECT creates a text field as a child of fields
  (define (add-tsuro-field! letter)
    ; TextField Event -> Void
    ; EFFECT connect the specified char in f with this letter
    (define (letter-callback f e)
      (connect! (send f get-text) letter))
    ; Container -> Void
    ; EFFECT create an option field as a child of p
    (define (option-maker p)
      (new text-field$ [parent p] [callback letter-callback]))
      (new labeled-option$ [parent fields]
        [label (format "~a: " letter)]
        [option option-maker]))
  (define field-gui ; create all text entry fields
    (for/hash ([a TILE-NODES])
      (values a (send (add-tsuro-field! a) get-option))))

  ; CODE GENERATION
  (define-elaborator this
    '#(new tsuro% [edges '#,(send this get-pairs)]))

Figure 3: Example Editor for Tsuro Tile
these containers work with the \texttt{widget} editor type. Each child has a super class that supplies its drawing and event-handling methods.

To illustrate these abstraction and composition mechanisms, figure 3 presents the implementation of the Tsuro tile extension. The purpose of this interactive syntax is to permit the programmer to insert a graphical image of the tile where an expression is expected. The construction and maintenance of this tile demands a capability for connecting and re-connecting the entry points of the tile, as well as for displaying the current state of the connections both graphically and as text.

Rather than implementing the \texttt{draw} and \texttt{on-event} methods directly, the Tsuro syntax relies on container classes (lines 4, 18–38) to manage layout and events. These labels and fields (lines 22–42) are provided by the standard library and can draw themselves. The \texttt{tsuro-picture} editor (lines 18–20) works with \texttt{trace-player} to provide drawing and event handling functionality.

The field and label sub-editors serve similar purposes; they both render text. The field editor, however, also handles user interaction, while the label one does not. These editors use the \texttt{text} and \texttt{focus} interactive-syntax mixins from the standard library; see figure 4. The \texttt{text} mixin (lines 1–5) handles both the text portion of the editor’s state and drawing directly. The \texttt{focus} mixin (lines 7–10) handles user interaction. Finally, the \texttt{text-field} editor (lines 12–14) combines the two mixins and applies them to the \texttt{widget} base.

The code elaborator (lines 44–46) in figure 3 turns \texttt{pairs}, the state of the extension, into a purely run time object, called \texttt{tsuro}, using the traditional syntax extension mechanism. This object contains the edges of connections for each tile stored as a hash table for the run-time phase.

Importantly, developers may compose interactive-syntax extensions. Thus, for example, an instance of \texttt{tsuro-tile} works with the editor for the full Tsuro board. Each tile in the board is stored directly in the board editor, renders itself in the board’s GUI context, and reacts to events flowing down from this container.

The GUI classes that make up these editor definitions are \textit{not} those of the standard Racket widget library. Rather, they are custom widget definitions that can be embedded directly in program text. Chapter 7 returns to the discussion of these problems.
; Mixin for drawing text in an editor
(define-interactive-syntax-mixin text$$
  (super-new)
  (define-state text "")
  (define/augment (draw dc) ...))

; Mixin for basic user interaction
(define-interactive-syntax-mixin focus$$
  (super-new)
  (define/augment (on-event event) ...))

; A text field widget
(define-interactive-syntax text-field$ (focus$$ (text$$ widget$))
  (super-new))

Figure 4: Field editor using mixins
A PLETHORA OF EXAMPLES

The Tsuro-specific extensions from chapter 2 illustrate two aspects of programming with visual and interactive-syntax extensions. First, the interactive composition of visual and textual code can obviously express ideas better than just text (or just pictures). In a sense, this first insight is not surprising. Like English, many natural languages come with the idiom that “a picture is worth a thousand words.” In short, interactive syntax combined with text is clearly useful.

Second, the implementation sketch demonstrates the ease of developing such interactive extensions. The effort looks eminently reasonable in the context of a prototype, especially since the essential code of this particular example can, in principle, be shared between the GUI interface to Tsuro and the unit test suites. That is, the mechanism also seems to be easily usable.

Naturally, a single example cannot serve as the basis of an evaluation. A truly proper evaluation of this new language feature must demonstrate its expressive power with a number of distinct cases. Additionally, it must show that the effort remains reasonable across this spectrum of examples. This chapter starts with a list of inspirational sources: numerous textbook illustrations of algorithms with diagrams, pictorial illustrations in standards such as RFCs, and ASCII diagrams in code repositories. The second section surveys a range of uses and implementations of those uses, with an emphasis on where and how interactive syntax can be deployed. The final two sections present two cases in some depth.

5.1 EXAMPLES OF DIAGRAM DOCUMENTATION

Textbooks, documentation, source code inspection, and practical experience all motivate the idea of interactive-syntax extensions.

- Tree Algorithms—Every standard algorithms book [Cormen et al. 2009] and every tree automata monograph [Comon et al. 2007] comes with many diagrams to describe tree manipulations. Programmers often include ASCII diagrams of trees in comments to document complex code.¹ These diagrams contain concrete trees and depict abstract tree transformations.

- Matrix—Astute programmers format matrix-manipulation code to reflect literal matrices when possible.² Mathematical program-

¹ https://git.musl-libc.org/cgit/musl/tree/src/search/tsearch.c?id=v1.1.21
² http://www.opengl-tutorial.org/
ming books depict matrices as rectangles in otherwise linear text [Fourer et al. 2002].

• **File System Data Structures**—Any systems course that covers the inode file representation describes it with box-and-pointer diagrams. Likewise, source code for these data structures frequently include ASCII sketches of these diagrams.

• **TCP**—RFC-793 [Postel 1981] for TCP lays out the format of messages via a table-shaped diagram. Each row represents a 32-bit word that is split into four 8-bit octets.

• **Pictures as Bindings**—Many visual programming environments, such as Game Maker [Overmars 2004], allow developers to lay out their programs as actors placed on a spatial grid. Actors are depicted as pictorial avatars and the code defining each actor’s behavior refers to other actors using avatars. In other words, pictures act as the variable names referencing objects in this environment.

• **Video Editors**—Video editing is predominantly done via non-linear, graphical editors. Such purely graphical editors tend to force people to perform manually repetitive tasks.

• **Circuits**—Circuits are naturally described graphically. Reviewers might be familiar with Tikz and CircuitTikz, two \LaTeX{} libraries for drawing diagrams and specifically circuit diagrams. Coding diagrams in these languages is rather painful, though; manipulating them afterwards to put them into the proper place within a paper can also pose challenges.

  Electrical engineers code circuits in the domain-specific simulation language SPICE [Vogt et al. 2019] or hardware description languages such as Xilinx ISE. While both come with tools to edit circuits graphically, engineers cannot mix and match textual and graphical part definitions.

### 5.2 The Expressive Power of Interactive Syntax

I have all implemented the examples from the previous section with interactive syntax in racket/visr. Doing so yields easily readable code and several insights on the expressive power of mixing visual and textual syntax. Here I present a classification of the linguistic roles that these extensions play within code. Figure 5 provides a concise overview. The first column lists the name of the example, the second the role that interactive syntax plays. The third column reports the number of lines of code needed for these extensions.

---

3 https://www.youtube.com/watch?v=tMVj22EWg6A
<table>
<thead>
<tr>
<th>Name</th>
<th>Elements</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tree Algorithms</td>
<td>Data Literal, Pattern Matching, Template</td>
<td>353</td>
</tr>
<tr>
<td>Matrix</td>
<td>Data Literal, Template</td>
<td>175</td>
</tr>
<tr>
<td>File System Data</td>
<td>Data Literal, Other Binding</td>
<td>178</td>
</tr>
<tr>
<td>TCP Parser</td>
<td>Data Literal, Pattern Matching, Template</td>
<td>98</td>
</tr>
<tr>
<td>Pictures As Bindings</td>
<td>Other Binding</td>
<td>88</td>
</tr>
<tr>
<td>Video Editor</td>
<td>Data Literal, Template</td>
<td>80</td>
</tr>
<tr>
<td>Circuit Editor</td>
<td>Data Literal</td>
<td>307</td>
</tr>
<tr>
<td>Tsuro</td>
<td>Data Literal, Template</td>
<td>408</td>
</tr>
<tr>
<td>Form Builder</td>
<td>Data Literal, Template, Meta Binding</td>
<td>119</td>
</tr>
</tbody>
</table>

Figure 5: Attributes of the Worked Language Extensions

- **Data Literal**—The simplest role is that of a data literal. In this role, developers interact with the syntax only to describe complex, often geometric, pieces of data; the elaborator tends to translate these editors into structures or objects. As the Tsuro examples in the introduction point out, data-literal forms of interactive syntax can be replaced with a lot of text—at the cost of reduced readability.

- **Template**—The template’s role generalizes data literals. Instead of creating plain data, a developer inserts code into text fields of these instances. The Tsuro board in the introduction and the tree in figure 7 are examples of such templates. The templates build a board and tree, respectively, using pattern variables embedded in an editor.

- **Pattern Matching**—Some languages emphasize algebraic pattern matching, and interactive syntax can greatly enhance the message that a pattern expresses. In this context, a developer fills an editor with pattern variables, and the code generator synthesizes a pattern from the visual parts and these pattern variables. In this role, pattern-matching editors serve as binding constructs.

- **Meta Binding**—Since syntax extension is a form of meta programming, interactive syntax naturally plays a meta-programming role, too. I refer to this role as meta-binding in figure 5. Here editors are used to construct new types of syntax for racket/visr, they can generate both graphical and textual syntax extensions. Section 5.4 demonstrates this idea with form builders.

- **Other Binding**—Finally, editors can play the role of a binding form. This role allows multiple editors to “talk” to each other. The inode data structure supplies an example of this kind.
5.3 **In Depth: Red-Black Trees**

When programmers explain tree algorithms, they frequently describe the essential ideas with diagrams. Often these diagrams make it into the library documentation and the programmer who maintains the code has to go back and forth between the code and the diagrams in the documentation. A poor man’s fix is to render such diagrams as ASCII art.\(^4\)

The balancing algorithm for red-black trees [Bayer 1972] illustrates this kind of work particularly well. Figure 6 shows a code snippet from a tree-manipulation library in Racket. The snippet depicts a function for balancing red-black trees using pattern matching. The comment block (lines 1–9) makes up the internal ASCII-art documentation of the functionality, while the code itself (lines 10–16) is written with Racket’s expressive pattern-matching construct.

An interactive-syntax extension empowers the developers to express the algorithm directly as a diagram, which guarantees that the diagram and the code are always in sync. The key point is that interactive syntax can show up in the *pattern* part of a *match* expression as well as in the *template* part, both situated within ordinary program text.

A look at figure 7 makes this point for the red-black tree balance algorithm. The *match* expression consists of two clauses, but only the first one matters for the current discussion.

The *or* pattern combines several sub-patterns; if any one of them matches, the pattern matches the given tree *t*. This example uses four sub-patterns, each expressed with editors; each represents one of the four possible situations in which a balance must take place. The situation should remind the reader of the diagram in Okasaki’s functional implementation [Okasaki 1999], which uses the same four

---

\(^4\) [https://blog.regehr.org/archives/1653](https://blog.regehr.org/archives/1653)
trees on the second page of his paper. The four sub-patterns name nodes—x, y, and z—and subtrees—A, B, C, and D—with consistent sets of pattern variables.

The template—on the second line—refers to these pieces of the pattern. It is also an editor and shows how the nodes and sub-trees are put into a different position. The resulting tree is clearly balanced relative to the matched subtrees.

In sum, the code consists of four input patterns in visual form that map to the same output pattern. Any programmer who opens this file in an IDE that renders the editors visually will immediately understand the relationship between the input tree shapes and the output tree—plus the connection to the published paper.

5.4 IN DEPTH: FORM BUILDERS

The presented examples generate only run-time code, and they exclusively focus on patterns and data structures. Interactive-syntext extensions, however, can also generate compile-time and even edit-time code in Racket. This means that interactive-syntext extensions can impose both domain-specific validation requirements and provide the tools to enforce them.

As an example, consider embedding table-like forms into code. To make this idea truly concrete, imagine managing a large introductory course, with several hundred students and a staff of a few dozen teaching assistants. The course coordinator must log information for
each student and staff member. Furthermore, each role requires different information, e.g., each student gets a grade, while staff members have grading assignments. To manage all this information, the course coordinator can use interactive syntax to create information forms. Rather than making forms directly, the coordinator creates a form builder with interactive syntax to generate each type of form. More generally, in this scenario interactive syntax is used to make new types of interactive syntax.

Normally, forms act like the Tsuro editor and elaborate to tables containing their contents. These tables can act as dictionaries used directly in source programs or as SQL statements that insert data into a form-specific database.

Programmers can add additional constraints to fields in those forms. Forms that do not meet these constraints are considered syntax errors. For example, here is a small program with an incorrectly filled grade form:

```
Bad syntax in student-form$: Invalid "Student ID", expected an: id?,
got: "Bob Smith"
```

The “Student ID” field expects an identification number, but the user put in a name rather than an ID. Because forms may be created with checks that ensure correct data is entered, feedback happens during edit time. As an aside, note the “Total Score” pseudo-field in this form, which displays a result, but is not a text field. In addition to traditional (text) fields, forms can contain any arbitrary interactive-syntax extension. For example, the “Total Score” area sums the result of “Problem 1” and “Problem 2”.

Rather than creating form editors textually, the coordinator can use a graphical form builder from a library of interactive syntax constructs to create new types of forms. The editor below is an example of this meta-form. Using an IDE action, the coordinator instantiated this form editor and used the text field labeled define-form to give a name to all its instances. As displayed, this meta-editor already specifies a
list of fields: "Student ID", "Total Score", "Problem 1", "Problem 1 Comments", "Problem 2", and "Problem 2 Comments". Each field comes with a button, with which the coordinator can remove the field from the form. Additionally, each field contains a reference to either an editor or a predicate. If an editor is provided, say total$, then the form uses it instead of the default text field. This enables custom sub-editors to display information based on other fields. Likewise, if a predicate is provided, it handles the validation of entries in the corresponding text field.

At the bottom is a text field plus a button; it permits the coordinator to add a new field to student-assign2$. Indeed, once a field is deleted or added, every instance of this student-form in the program is updated to match its meta-definition. For example, if the coordinator were to add a "Problem 3" label, a correspondingly labeled, blank text field would show up in both of the editors in the list on the next line.

In addition to defining new form types in the program, the meta-editor introduces a companion SQL table to the program. The table’s schema is specific to the introduced form type, which allows users to add form instances into the table. The student-assign2$ table, for example, expects five fields: "Student ID", "Problem 1", "Problem 1 Comments", "Problem 2", and "Problem 2 Comments". The "Total Score" field is absent because the total$ editor does not provide an SQL schema.

Figure 8 presents the implementation of the form builder syntax. This interactive-syntax extension is like a graphical GUI editor in that it generates another interactive-syntax extension. The edit-time code, such as the state and view, is similar to the Tsuro editor and hence of little interest. Its elaborator contains compile-time code that generates more code that runs at compile time, run time, and edit time. First, at compile time it generates identifiers used throughout the elaborator (lines 4-9, compile time). Second, the elaborator defines an SQL schema for form instances at run time (lines 11-12, run time). Finally it synthesizes the edit-time code for the form itself (lines 13-17, edit time).

The generated interactive-syntax extension looks deceivingly simple:

- The new interactive-syntax class, named name as specified in the state of form-builder$, inherits from the table-base$ superclass (lines 13). This use of inheritance illustrates a common use of generated interactive-syntax definitions. Rather than putting the entire implementation for an editor in the template of the elaborator, I factor most of it out into an external class. Besides separating common code from elaborator-specific one, it also generates significantly smaller code than a full-fledged implementation class, which significantly improves compile-time and edit-time performance. The table-base$ class itself is rather pedestrian GUI code.
The elaborator of the generated interactive-syntax simply expands to a dictionary with the form’s field names and values.

While generating interactive-syntax extensions from interactive-syntax extensions sounds complicated, the decades-old precedent of syntax extension design should help the reader understand the incredible power in such meta-capabilities.

In short, interactive visual syntax can be used to introduce new interactive visual syntax. Here a general form-generating interactive syntax construct is instantiated for creating assignment specific forms. This meta-editor adds another editor type to the program. At some point, of course, the process has to switch back to text, which is where the form extension itself is defined.
The implementation of an interactive-syntax extension mechanism poses four challenges, besides adding the syntactic forms of section 4.3. First, the language’s syntax must be augmented with a textual editor form to house instances of these extensions. Second, the language’s semantics must be extended to support the execution of code at edit time. Third, the language must include some mechanism for interactive-syntax elaboration. Finally, it demands the construction of plugins for graphical IDEs so that they use these extended semantics to interpret the textual editor form as a visual and interactive graphical user interface. Any language tool chain that can accommodate these four components can also fully support interactive-syntax extensions.

This chapter first describes a general implementation approach that mostly realizes the design desiderata and suggests abstract solutions to these four challenges. It then explains how the application of this approach to Racket yields a reasonably robust prototype[^1] that connects to the DrRacket IDE.

### 6.1 Ingredients for an Interactive-Syntax System

The four efforts required to implement visual interactive syntax in a language are: editor syntax, edit-time semantics, editor elaboration, and IDE integration.

*The Editor Form*

For a programming language to support interactive-syntax extensions, implementers of that language must extend its syntax to include a textual representation for editors. This textual representation is also how an editor is stored on the developer’s file system. Additionally, using a purely textual representation allows developers to edit their code in any environment, such as plain text editors, not just interactive-syntax specific ones.

This textual extension must contain three pieces of information: an editor’s state, a pointer to a means of converting that state into a graphical editor, and a pointer to a means of converting that state into run-time code. It can be added either to the language’s core implementation or as an external language extension. Many common languages, such as C, already allow for small language extensions like

[^1]: https://github.com/videolang/interactive-syntax
this. Languages that don’t, such as Java, may still have tools that can emulate language extensions to a limited degree.

As an example, an instance of the form builder from figure 8 may have this textual representation:

```rkt
$editor {
  binding: ["lib/form-builder.rkt", "form-builder$"],
  state: {
    name: "person$",
    keys: ["Name" ,"Age"]
  }
}
```

Here, the binding tag refers to the module and name of an interactive-syntax binding, which serves the dual purpose of converting state into a graphical element and into elaborated code. The rest of the syntax contains the editor’s state as a hash table. Due to this design choice, a plain text editor, such as Emacs or Vim, simply displays this text when a developer opens a module that contains interactive syntax. IDEs with support for interactive syntax can display the editors as mini GUIs embedded in program text, using the information stored in the binding field.

The interpretation of editor syntax is analogous to closures. Like a closure, an editor combines a code pointer and its current state into a new kind of value. The code pointer, found in the binding keyword in the example above, refers to the define-interactive-syntax definition that the editor instantiates. The state component records those aspects of the editor’s state that this definition specifies as persistent. Together these two pieces suffice to fully re-instantiate the editor as a graphical element in IDEs that can interpret these “closures” at edit time.

**Edit-time semantics**

The second component required for interactive-syntax extensions is semantics for edit-time code. This semantics is distinct from already existing semantics for compile-time and run-time code. Furthermore, it must be possible to interleave edit-time code with other forms of code in a program’s body and to formulate such edit-time code in the same language as run-time code or compile-time code. Finally, while not strictly necessary, the language should enforce a level of isolation on effects among these phases to allow a clean separate compilation of modules in the presence of co-mingled elements [Flatt 2002].

Two syntactic forms must link run-time code and edit-time code. The first form simply allows edit-time code to be spliced into run time code. The second form must create bindings in run-time code, that refer to extensions defined at edit-time. As with the syntax extensions,
this semantics can be either built into a language’s core, or it can be added with an external preprocessor.

**Editor Elaboration**

The language also needs an expander to turn each editor form into its elaborated code. This expander must additionally be able to evaluate user code including the definition of new types of interactive-syntax. An expander can use its host language’s normal meta-programming facilities. However, these facilities must implement the one-and-a-half pass approach discussed in chapter 4 to accommodate meta-edit-time programming.

Expanding an editor into code happens in five steps. First, the expander must recognize editor syntax. Second, it must deserialize an editor’s state. Third, the expander must locate the editor’s elaborator. Fourth, the expander invokes the elaborator with the editor’s state. Finally, it splices the generated code into the program body.

**Cooperating With an IDE**

An IDE that supports visual and interactive syntax must cooperate with the language to run the edit-time code of interactive-syntax extensions and to connect this code to its own graphical context. An implementation can either modify the IDE directly or via a plugin. If an IDE supports interactive-syntax extensions through a plugin, that plugin must not require each extension to install its own plugin.

Enabling an IDE to interpret editors demands three kinds of extensions. First, the IDE must recognize instances of interactive-syntax extensions so that it supports UI actions for the insertion of instances into code and for updating existing editors if their underlying definition changes. Second, the IDE must supply a graphical context to editors so that they can render themselves and receive relevant UI events. Third, the IDE must execute edit-time semantics in a reasonably sandboxed environment.

The code that comprises an interactive-syntax extension is fundamentally user code. As such, the IDE must sandbox these extensions. This sandbox is similar to the traditional environment that an operating system may use for processes or a web browser may use for its tabs. For example, a sandbox may prevent an extension from creating or modifying files without some file system permission. This sandbox environment must also allow an editor to gracefully fail, reverting back to some default rendering or even a variant of its textual representation.
6.2 A Prototype Using Racket and DrRacket

As a proof of concept, I have constructed racket/visr, a prototype of interactive-syntax extensions for the Racket language [Flatt and PLT 2010] and DrRacket IDE [Findler et al. 2002].

The Editor Form

First, an editor’s textual representation is composed of binding information and state syntax. Concretely, an editor is represented in text similar to the form in the preceding subsection.

Making this form valid syntax requires a change to Racket’s reader. The extended reader generates a valid syntax object with a known (macro) interpretation. Racket’s reader is extensible, meaning that interactive syntax can, in principle, work with any Racket-based language [Felleisen et al. 2018] that also uses Racket’s reader extensions.

Edit-time semantics

Second, while Racket already supports a hierarchy of compile-time phases (for syntax extensions that generate syntax extensions), it has no mechanism for adding a new phase. Since I wish to demonstrate that interactive-syntatx extensions can be added to a language without changing the underlying virtual machine or interpreter, the prototype employs a surprisingly robust work-around. Similar work-arounds may exist for other languages too, though it generally does require the implementer to create some sort of an edit phase.

Interactive-syntax extensions are implemented as syntax extensions. They elaborate constructs such as define-interactive-syntax, editors, and begin-for-interactive-syntax into a mix of further (plain) syntax extensions and submodules.

Figure 9 shows the elaboration of an interactive-syntax definition. The module in the top half consists of three pieces: the definition of an interactive-syntax extension, an editor of this extension (the #f denotes “locally defined,” the simple$ points to the definition; the editor has no state), and a simplistic edit-time test of this definition. The module at the bottom is (approximately) the transformation of the module at the top into Racket code.

In the expanded program, all edit-time code is placed into a single edit submodule [Flatt 2013]. This new submodule is inserted at the bottom of the expansion module. The definition of the interactive-syntax extension is separated into two pieces (as indicated with code highlighting and indicies): the elaborator called simple$:elaborator, which exists at compile time, and the interactive-syntax class called simple$, which exists at edit time. Recall that the elaborator translates the state of an editor into run-time code; the interactive-syntax class
Elaboration of an Interactive-Syntax Extension

Figure 9: Elaboration of an Interactive-Syntax Extension
inherits and implements the edit-time interaction functionality for the syntax extension. As for the textual editor form, its reference to the `simple$` interactive-syntax extension is refined to a reference to the elaborator; for edit time execution, the IDE plugin performs a separate name resolution. Finally, the test code in the `begin-for-interactive-syntax` block is also moved into the `edit` submodule.

Placing the edit-time code into a separate submodule permits the runtime system to distinguish between editor-specific code and general-purpose program code. In particular, it ensures that the runtime system can execute the editor portion of an interactive-syntax extension independently of its host module. Indeed, the runtime system realizes this goal by merely requiring the `edit` submodule and thus obtaining the provided `simple$` interactive-syntax class, which implements the GUI interactions. By contrast, the generated run-time code of an editor must remain subject to the host module’s scope.

**Editor Elaboration**

Third, editor elaboration is a straightforward use of Racket’s macro expander. The Racket parser converts the `#editor{...}` form in the source into traditional Racket syntax containing a `#%editor` macro. The `#%editor` macro (figure 10) finds the elaborator (lines 3–5), deserializes the state (lines 6–7), and expands to the elaborator with the deserialized state (line 8). From here, the macro expander places the residual code back into the program body.

**Cooperating With an IDE**

Finally, the prototype exploits DrRacket’s plug-in API for the event handling [Findler and PLT 2010] and its Cairo-based drawing API for editor rendering. A specially designed plug-in connects interactive-syntax extensions to the IDE. It inserts menu entries that programmers can use to instantiate interactive-syntax extensions and insert their

```
1 (define-syntax #%editor
2   (syntax-parser
3     [(_ (module name) body)
4       (define/syntax-parse elaborator
5         (forge-identifier #'module #'name))
6       (define/syntax-parse state
7          (deserialize-state #'body))
8       #'(elaborator state)]))
```

Figure 10: Implementation of the `#%editor` macro
instances into specific points in the code (figure 11). The plug-in also assists with saving and retrieving modules that contain editors. When a developer saves a file, the plugin serializes all instances of interactive syntax extensions into $editor{...}$ blocks; conversely, when a developer opens such a module, it uses the language’s parser to scan the file for $editor{...}$ blocks and informs the IDE about them.

Technically, the prototype relies on Racket’s GUI toolbox and sandboxes mechanism [Flatt et al. 2010]. Specifically, the Racket evaluator provides controlled channels for sandboxed namespaces to connect to the rest of the Racket runtime system. The Racket GUI toolbox already supports graphics within textual programs via the snip API. DrRacket supplies a drawing context to snips and passes user events to them. However, snips are extra-linguistic. They are thus IDE-specific elements and not elements of the language the programmer edits. They were DrRacket’s first attempt at mixing graphical and textual programming, but made any file that used them unreadable outside of DrRacket. The prototype bridges the gap between these two GUI elements so that editors remain language forms, yet connect to the IDE smoothly.

The prototype accommodates failures with a simple fallback editor GUI. If a developer were to inject a typo into an $editor{...}$ via Emacs or were to move the file that contains an interactive-syntax extension, the prototype does not crash. Instead it hands control to a form editor, which displays a small default GUI whose fields show the editor’s binding and state information. The following is this fallback editor for the form builder:
The module is found in "lib/form-builder.rkt", which provides form-builder$ as an identifier. The editor has two state fields: name and keys. Clicking on the initialize button tells the runtime system to make another attempt at re-initializing the editor with these values.
Unfortunately, the racket/visr prototype falls short of the design requirements laid out in chapter 3. The chosen language, Racket, currently fails the prototype in two major ways:

1. its GUI framework is not powerful enough, requiring developers to use separate implementations for their application view and interactive-syntax, and

2. limitations in its macro hygiene system end up making some extensions difficult or impossible to create.

While the second problem can be mitigated by admitting temporary shortcoming with respect to hygiene, the limitations of Racket’s GUI framework turn out to be a show stopper. In short, racket/visr fatally violates the third design requirement of chapter 4.

Most notably, visual syntax extensions in racket/visr cannot make use of the existing Racket GUI library. Instead visual syntax extensions must use a custom GUI library. This major limitation means that while racket/visr is useful for communicating ideas and thoughts via hybrid code, it is ultimately not usable.

This technical problem is due to a division of Racket’s GUI library. This GUI library comes in two parts: an operating-system widgets layer and a code-editor layer. Code editors can be placed into both widgets and other code editors. Widgets, in contrast, can only be placed in other widgets. The design of racket/visr, however, necessitates embedding widgets in code-editors.

To work around this, I created a widget library that attempts to parody the Racket widget library inside of code editors. In principle, programs written for one library can be easily ported to the other. In practice though, porting these programs from one library to the other is a significant effort. As a result, to make use of interactive-syntax extensions, programmers are forced to develop two implementations of their GUI.

An unfortunate side effect of using a custom interactive-syntax GUI framework is that this custom framework is inevitably less performant than its user-facing counterpart. This means that while developers are able to read code with that uses interactive-syntax extensions, their ability to write new code is severely limited.

Document Object Model(DOM) based GUI toolkits such as those found in web browsers would be the perfect environment for mixing 1

---

1 By avoiding hygiene, interactive syntax does inherit the problems that plague non-hygienic macro expanders, such as the one in ClojureScript.
widgets and text. Indeed, its entire design revolves around making this rendering both efficient and aesthetically pleasing. Sadly, Racket’s web variant is currently impractical. Until this changes, \texttt{racket/visr} remains useful, but unusable. The next part of this dissertation lays the framework for ultimately deciding whether a usable implementation exists.
Part II

A USABLE IMPLEMENTATION FOR INTERACTIVE-SYNTAX
In order to make interactive-syntax usable, I must first define what it means for a language construct to be usable. I propose the following three criteria for evaluating the usability of a language with an interactive-syntax extension mechanism:

1. comprehension of an existing editor, or readability;
2. use of existing extension types in code, or writability; and
3. creation of new extension types, or extensibility.

A mechanism for extending programming languages with interactive-syntax is only usable if it satisfies all three criteria.

Note the difference between usability and usefulness discussed in Chapter 1. A language that satisfied only the readability criteria can still be useful as a mechanism for communicating algorithms. Likewise a language can satisfy all of the above criteria and still not be useful.

The ultimate goal is to validate the feasibility of creating a usable language with interactive-syntax extensions. Observe that a Document Object Model (DOM) based GUI system is made to accommodate a hybrid textual-visual approach. Furthermore, a language with access to the DOM encourages a syntax-extension mechanism. These two criteria naturally suggest using ClojureScript [McGranaghan 2011] as a host. I extend ClojureScript with core.visr, a plain ClojureScript file, for interactive-syntax extensions.

Unfortunately, ClojureScript does not come equipped with an IDE, so I also must create an IDE that renders interactive-syntax extensions. The elIDE IDE is a web-based IDE capable of rendering ClojureScript programs that use interactive-syntax extensions. Programmers can substitute using elIDE with any IDE. However if the replacement cannot render interactive-syntax extensions they must edit the textual representation of their code.

With this infrastructure in place, I can perform user-facing experiments and case studies to demonstrate that ClojureScript, core.visr, and elIDE are usable.

Chapter 9 lays out the design of this usable interactive-syntax system. Using this design, chapter 10 compares the implementation of the Tsuro example in racket/visr with a new one in core.visr. Chapter 11 provides details on how to implement core.visr. Next, chapter 12 and chapter 13, through the use of user facing studies, validate that interactive-syntax extensions are usable. Finally, chapter 14 compares racket/visr in DrRacket with core.visr in elIDE.

1 https://dom.spec.whatwg.org/
DESIGN

To start, I extend a language with support for interactive-syntax extensions, core.visr. Then, I create a supporting IDE, based on the Document Object Model (DOM), for that language.

The purpose of core.visr is to bring interactive-syntax to a programming language with a GUI toolkit for the browser, and more generally, anything that can render the DOM. A web browser is the ideal environment for a programming system that supports interactive-syntax extensions. The web is already filled with documents containing a mixture of textual and interactive elements. These documents are both readable and writable by users, such as a login form or even a small video editor. Web browsers have been highly optimized to give users a smooth and responsive UI. Additionally, the design of HTML and the DOM are well suited for displaying mixed textual and graphical content. Therefore, interactive-syntax naturally fits into this space.

Given a DOM-based environment, ClojureScript is a suitable host language for adding an interactive-syntax extension mechanism. Like Racket and Clojure, ClojureScript is a Lisp with macro support for extending the language. ClojureScript also has a bootstrapped compiler, meaning that development for it can happen right in the browser and thus a browser-based IDE. Finally, because ClojureScript compiles directly to JavaScript, it is compatible with existing JavaScript libraries, frameworks, and tools.

This chapter outlines the design of core.visr and its new IDE, elIDE.

9.1 ARCHITECTURE OVERVIEW

Figure 12 shows a diagram of the architecture of core.visr and elIDE. The top half of the figure lays out elIDE while the bottom half lays out the architecture for core.visr itself. The thin arrows represent communication between components while the thick arrows represent events triggered by some user action.

The main component for the IDE is a text editor powered by CodeMirror. The programs are stored in a filesystem powered by BrowserFS [Powers et al. 2017]. Arbitrary HTML components can be placed in this IDE using React. Finally, the IDE can run ClojureScript programs compiled with the ClojureScript compiler and evaluated

---

1 https://codemirror.net/
2 https://reactjs.org/
Figure 12: An Interactive-Syntax Language and its IDE in a DOM Architecture

using the Stopify [Baxter et al. 2018] environment. All interactive-syntax instances constantly run in the background as programmers edit code. A core.visr specific reader handles this task, again using ClojureScript and Stopify for evaluation.

The language is a vanilla ClojureScript runtime bundled with a bootstrapped ClojureScript compiler and a small library for core.visr. For space efficiency, ClojureScript programs don’t normally run with a bootstrapped compiler. While core.visr doesn’t need the bootstrapped compiler to run programs using interactive-syntax extensions, it is needed for visual renderings of code.

9.2 Evaluating ClojureScript in a Browser

While ClojureScript’s macro expander is simple, it is powerful enough to support interactive-syntax extensions. ClojureScript’s metadata construct allows programs to embed readable and evaluated blocks of data from the program itself. Designed akin to reader macros, programmers can attach metadata to any expression in the code. For example, metadata is frequently used to embed compiler hints in a program. Interactive-syntax extensions can use this metadata to embed an object’s edit-time state.

Unfortunately, ClojureScript is not a perfect fit. First, ClojureScript requires macro definitions to be placed in a separate file from the rest of the code. Therefore definitions of interactive-syntax extensions in ClojureScript that use macros must also need be in their own file. Second, ClojureScript relies on the browser for all of its sandboxing. For core.visr, sandboxing is used between interactive extensions and
the browser, between extensions and the running code, and between individually running extensions. That is, the architecture sandboxes editors from the program’s run-time semantics, but does not sandbox editors from each other.

Stopify acts as a sandbox for interactive-syntax editors. In particular, Stopify supports two essential features over the vanilla ClojureScript evaluator. First, it allows the IDE to pause running programs. That is, misbehaving interactive-syntax extensions will not lock up the entire application. Second, it provides a sandbox environment, separate from the IDE, for the extensions to run in. This allows programmers to run their programs without colliding with the IDE’s current internal state.

Finally, SystemJS\(^3\) enables visual programs to link to code written in vanilla JavaScript and WebAssembly code.\(^4\)

9.3 The elIDE IDE

Rather than building a full IDE from scratch, core.visr uses the CodeMirror development environment for text editing. CodeMirror allows embedding arbitrary HTML components within the text field, primarily for the purpose of providing documentation from within the IDE and highlighting errors. The elIDE IDE uses this mechanism to add interactive-syntax extensions from core.visr to source.

The interactive-syntax extensions themselves come from the program being edited. This is the only way for core.visr to send information back to the IDE. Each interactive-syntax extension uses React to manage state and visualization.

While ClojureScript programs can consist of multiple files, web browsers do not give webpages direct access to a user’s filesystem. To resolve this issue, elIDE uses BrowserFS to create a virtual filesystem stored in the user’s local storage.

---

\(^3\) https://github.com/systemjs/systemjs
\(^4\) A database of packages formatted to work with the IDE can be found at: https://github.com/LeifAndersen/visr-deps.
Chapter 2 motivates the usefulness of a hybrid language through the use of a Tsuro specific interactive-syntax extension. Therefore, it is fitting to return to the Tsuro extension for core.visr to see how interactive syntax works in this context. This serves four purposes. First, it shows how users interact with extensions in core.visr. Second, it shows how core.visr realizes the components of an interactive-syntax extension. Third, it illustrates how well core.visr implements the programming workflows of chapter 3. Finally, it provides an opportunity to compare this prototype to its Racket counterpart, including the code examples run in racket/visr.

10.1 The Tsuro Editor

Figure 13 shows an editor of the Tsuro tile extension. It depicts the visual view on the left and the textual representation on the right. The text side contains a reference to the specific extension used (tsuro.core/Tile) plus the state of the extension. When a programmer modifies the code in the visual view, its textual counterpart immediately updates to reflect this change, and vice versa.

This tile editor stores the tile information as a set of node pairs from 'A' to 'H.' In contrast to the racket/visr implementation this editor lacks explicit text fields for each connection, for two reasons. First, this extension allows users to directly drag and drop each path significantly more fluidly than in racket/visr. Second, the side-by-side views of the textual and visual representation make explicit text fields superfluous; a programmer can edit the text directly. Of course, entering bad data as text into the state could prevent a proper
visualization, but once the error is fixed the visualization returns as well.

When put in a program, this extension elaborates to a runtime representation of this tile. Figure 14 shows the tile in a short unit-test. Lines 1–3 tell the run-time system where to find the interpretation of this particular extension, while the extension itself is on line 5, with the text view hidden.

10.2 THE COMPONENTS OF AN EXTENSION WITH core.visr

The definition of a visual ClojureScript extension specifies only an elaborate method and a render method:

```clojure
(defvisr Tile
  (elaborate [this] ...)
  (render [this] ...))
```

The render method serves the dual purpose of drawing a component and handling user input. This arrangement follows standard web-development practices, collapsing view and control. The this parameter passed to both methods contain the state. Changing the state programmatically is done by modifying this as a ClojureScript atom in the render method. The state has the same binding in compile-time, run-time and edit-time.

The elaborator method (elaborate) for the Tsuro example is shown in figure 15. The run-time representation for the tile is the set of
connected nodes. The state is read-only in the `elaborate` method\(^1\) and destructured on line 2. Because the run-time representation is stored directly as editor state, the elaborator evaluates to the destructured state, also on line 2.

The render method (`render`), also shown in figure 15, reuses the run-time GUI code for the Tsuro game. The render method itself is on lines 3–5, with lines 7–25 responsible for drawing the tile both at run-time and edit-time. Unlike the elaborator, the renderer gets the state wrapped in an atom. Line 4 performs the destructuring of the atom as well as initializing it to an empty set on the first run.\(^2\)

The renderer is a function from state to a data structure that encodes the HTML the user sees. Functions placed in the first position in a vector (e.g. `tile-view` on line 5) are treated as sub-components to be rendered. Likewise, keywords (e.g. `:div` on line 10) represent direct HTML tags. Finally the `:>` keyword acts as a foreign function interface (FFI) [Sexton 1988] by converting a plain React component into one the renderer can process.

Reading and modifying the editor state is done through interactions with the state atom. Unboxing the atom, through the `@` operator,

---

1 Changing the state in the elaborator would be analogous to having a compiler change the file containing a program’s source code.

2 For readers unfamiliar with Reagent in ClojureScript, cursor atoms return null when their value is uninitialized, which indicates that figure 15 must setup the state to an empty set (`#()`).

---

Figure 15: Implementation of the Tsuro Tile Editor
returns a purely functional encoding of the state (line 14). Updating the atom is done through calls to \texttt{reset!} and \texttt{swap!} (line 20). When the state atom changes, a publish–subscribe style watcher notices and updates the user’s view.

10.3 Programming Workflows

- **Auditing** Visual-interactive extensions written with \texttt{core.visr} allow users to view both the textual and visual variants of their code simultaneously. They will only see the textual variant when viewed in an IDE that does not support \texttt{core.visr}.

- **Creation** Users can press an Insert VISr button to instantiate a visual-interactive extension as an editor. Alternatively they can just type the starting text for an editor directly and the IDE will automatically detect it and render the visual component [Raskin 2000]. Modifying an editor’s textual or visual view mirrors plain text—it is stored in the IDE and converted to text either in real time.

- **Move and Copy** Programmers can cut, copy, and paste editors both within a single file and across multiple files. In fact, copy/paste works across entirely different IDEs, including ones that do not support editors at all. All of this is possible because editors are stored as human-readable text.

- **Running** Editors, as a graphical construct, exist only during edit-time, and therefore have no impact on a program’s execution. In fact, the \texttt{defvisr} form is simply a ClojureScript macro that expands to two function definitions.

- **Search and Replace** Normal IDE search and replace works on programs using editors. The only limitation is that it highlights the whole editor, rather than the particular portion of the editor in which the search-text is found. A small amount of engineering could rectify this for simple extensions with purely textual sub-editors. As of right now the search and replace dialog itself only accepts plain text, although in principle it could also accept something visual as its input.

10.4 Comparing the Tsuro Extensions

The obvious difference when comparing \texttt{racket/visr} to \texttt{core.visr} is the difference between four apparent ingredients in \texttt{racket/visr} as opposed to seemingly two in \texttt{core.visr}. In particular, the initial state and the state updater appear to have gone missing. In actuality these two components still exist, only implicitly. Whereas the Racket variant treated the renderer as a pure function, the ClojureScript variant
expects its renderer to perform a side-effecting update whenever the state changes. While this change may seem unappealing to functional purists, it follows the current ClojureScript conventions of putting any required state in an atom and to unbox that atom when state mutation is not required.

The `define-state` syntactic form allowed interactive-syntax to have multiple pieces of state. As a consequence of removing this capability, editors in `core.visr` can have only one piece of state, named as `this`. To compensate for this limitation, this one piece of state is usually a hash table mapping keywords to individual pieces of state. Again, this arrangement follows the ClojureScript convention of using data structures as a proxy for encoding semantic meaning that is interpreted at run-time. Of course all of this is happening with respect to the program’s edit-time. So another, ad-hoc, way to think of this is `core.visr` defines its state at edit-run-time, while `racket/visr` defines its state at edit-compile-time.

The implementation overhead for the Racket variant was roughly two to three times its ClojureScript counterpart, both in time and lines of code. In `racket/visr`, the Tsuro tile widget takes roughly 300 lines of code. While in `core.visr`, the entire implementation fits in 80 lines. The exact engineering time for each was not recorded for both widgets. However, both were implemented over the course of a few hours.

It is important to note that due to the small size of this example, this effort halving may not scale to large code bases. Also these metrics don’t account for time spent maintaining these editors.

The first real win for `core.visr` comes when using the Tsuro editors. On the one hand, the editors in Racket are clunky and extremely difficult to deal with. On the other hand, the ClojureScript counterpart is smooth and natural to use. Being able to manually drag and drop edges helps with usability.

The second win concerns the actual re-use of application-interface code in the interactive-syntax extension. This re-use is the key to making `core.visr` a truly usable language with interactive-syntax.
The implementation of the design in chapter 9 is fairly straightforward. It consists of 4,500 lines of code, with 1,400 lines dedicated to e1IDE specifically. While the prototype follows the architecture and design presented in figure 12, I have also experimented with alternatives. For example, I created a variant of core.visr that doesn’t use Stopify, and instead uses a custom evaluator.

For the most part, core.visr runs on vanilla ClojureScript and Stopify. Adding a few missing few missing features to make a DOM-based IDE with Stopify support took only a short time. In total these changes amount to about 100 lines of change to ClojureScript, and 25 lines to Stopify.

These changes fit into three broad categories. First, the ClojureScript evaluation API must follow an asynchronous call pattern. Second, ClojureScript’s evaluator must run in a sandboxed virtual environment. Finally, Stopify must support context switching. This section discusses these three changes.

### 11.1 Asynchronous API

The first change is for ClojureScript. While ClojureScript’s code evaluator is asynchronous by design, its module loader is completely synchronous. Unfortunately, JavaScript lacks the primitives to synchronize asynchronous code, which poses a problem when trying to mix the module loader with an inherently asynchronous filesystem.

For example, the following code requires ClojureScript’s evaluator to invoke its moddule loader:

```clojure
(ns example.core
 (:require [example.helper-lib]))
```

To evaluate example.core, the run-time must first obtain and evaluate example.helper-lib. If, however, the program text is not immediately available, the module loader has nothing to return.

Fortunately, there is no particular reason the module loader must be synchronized, as the rest of the evaluator is asynchronous. This whole situation can be rectified by changing the loader’s API to take a callback. In essence, this means a conversion to continuation-passing style [Fischer 1972; Reynolds 1972] for the evaluator’s use of the loader.

In the end, the load function has the following API:
(defn load-fn
  "{:name String :macros Bool #:path Path}
     Dictionary
     ( {:source String ...elided...} -> Void)
     -> Void"
  [{:keys [name macros path]} options cb]
  ...elided...)

That is, a function that takes three arguments: the first being a map containing three arguments, the second being an options argument, and finally a callback. Here, name is the name of the module as a symbol, example.helper-lib in the example. The macros argument is a boolean set to true if the module is being loaded for macro evaluation, false in the previous example. Finally, the path argument is the file location where the ClojureScript evaluator believes the file is, example/helper_lib in the previous example.

11.2 SANDBOX ENVIRONMENT

The second change is also for ClojureScript. Out of the box, ClojureScript runs everything in a namespace that is single, flat, and global. The expected way to refresh that namespace is to refresh the web-page or to restart the server. While fine for most apps, it is not acceptable for any app that evaluates user-provided code, such as an IDE.

An IDE requires two environments. The first environment is for the IDE itself. The second environment is the sandbox in which the IDE runs code. The code the IDE runs generally only has access to that second environment.

Stopify provides a solution to this problem. While the primary purpose of Stopify is to allow JavaScript evaluation that does not freeze the web-browser, it has a side benefit of providing a virtual environment for code execution. By redirecting compiled code to use this second environment, the IDE remains free to use the native JavaScript environment for its own state.

While it is possible to redirect the compiled code to use this virtual environment, the provided ClojureScript standard library still evaluates using the primary JavaScript environment. Therefore, ClojureScript must be equipped with an additional parameter used by the standard library to find the environment it should evaluate in. Once this parameter, *additional-core*, exists, core.visr sets it to the Stopify environment before evaluating user code.

11.3 CONTEXT SWITCHING

The second change leads to the third and final change, this time to Stopify. While the parameterizable global environment allows
core.visr to use the Stopify environment, the IDE still needs access to the unmodified ClojureScript environment. This is a side effect of the IDE itself running ClojureScript. If elIDE were written in vanilla JavaScript, for instance, this problem would not exist.

The *additional-core* parameter described earlier must be set before executing user code, and returned to its previous value before returning to IDE code. Stopify, however, makes this task difficult, as it intermixes the evaluation of IDE code with the evaluation of user code.

In essence, the parameter must be set whenever there is a context switch to user code and restored whenever control switches back to the IDE. Stopify has a callback for one direction, onYield, but not the other, onResume. Adding this second callback in the appropriate place allows elIDE to make both context switches when appropriate.
EVALUATING THE READABILITY AND WRITABILITY OF INTERACTIVE-SYNTAX

A user study is a good way to evaluate the usability of core.visr [Goodman et al. 2012; Ko et al. 2013; Lazar et al. 2017; Myers et al. 2016; Rubin and Chisnell 2008]. The study I designed provides participants with several programming related scenarios involving interactive syntax as well as starting code for those scenarios. It then asks those participants several multiple-choice and free-form questions about the scenario and the provided code. Finally the study asks participants to self-evaluate their programming expertise and their experience using interactive-syntax extensions.

For this study, the population is all programmers that could potentially use interactive-syntax extensions in their ClojureScript code. Enrollment in this study was open, and participants were recruited through announcements in various ClojureScript communities.

With this study, I aim to answer the following research questions:

- **RQ1** Are developers able to read programs that use interactive syntax? (readability)

- **RQ2** Are developers able to write programs using existing interactive-syntax extensions? (writability)

- **RQ3** Are developers able to create new types of interactive-syntax extensions for their programs? (extensibility)

The study described in this chapter aims to answer RQ1 and RQ2 specifically. The next chapter addresses RQ3. For context, chapter 5 show that the answers for racket/visr are “affirmative” for RQ1, “with limited success” for RQ2, and “negative” for RQ3.

12.1 APPROACH

My study asks a combination of multiple-choice and free-response questions about two programming scenarios. Each set of questions is about code that comes in a purely textual variant as well as one enriched with interactive-syntax extensions. The protocol randomly divides the participants into two groups. The first group gets the visual variant for the even questions and the textual variant for the odd questions. The second group gets the opposite, textual for even and visual for odd. Finally the order of the answers provided in the multiple choice questions are scrambled before being presented to
each participant, the questions themselves are provided in a fixed order.

While participants are asked each question, they can skip as many as they like. Additionally, participants are allowed to return later to amend their answers. Responses are tracked in real time, but only the final response to each question is evaluated. The response timestamps and partial responses are available.

Participants are given no introduction to core.visr before the study. Rather, questions in the study serve the dual role of teaching participants how to use interactive-syntax extensions. Participants who want to learn more about core.visr are directed to a blog post describing the basics of interactive-syntax and core.visr specifically.

The first scenario puts participants in the position of developing a small piece of a turn-based game played on a hexagonal grid. Tokens are placed on the grid and players take turns moving their token to an adjacent tile. When a player moves their token, they remove the previously occupied tile from the board. Players can only move to an unoccupied tile. The last player to move wins.

Figure 16 shows an example question, both its visual and textual variants. The textual version represents the board as a two-dimensional array. The layout follows a parallelogram shape where each entry corresponds to the state of one hexagonal space. An empty space is represented with \( :x \), an unoccupied tile is represented with \( :- \), and a player token is represented with a number. The visual version shows the hexagon board. The expected answer for this question is A.

The second scenario has subjects create a component to show a number between 0 and 7 on a 7-segment display. The component’s input, which is the number to display, is a 3 bit binary integer. The output is 7 bits, encoding each light’s on/off state. The free-response question for this scenario is shown in figure 17. In this example, participants are asked to use the display editor to implement a function that converts a number into a bit vector for the segments to display. The first segment is done for them, as well as test cases for the display. The expected result is a completed display that passes all of the given test cases.

The final part of the survey asks questions to gauge the participants level of experience with ClojureScript and programming in general. These questions also ask participants to rate their experience of using core.visr, and provide them a place to share any additional feedback. Figure 18 lists the questions asked.

At a high level, the multiple-choice questions are designed to test the readability of programs using interactive-syntax (RQ1), while the free form questions test the writability (RQ2). I may conclude that interactive syntax is readable if the majority of participants gets the expected answer to the multiple-choice questions. Furthermore, I can conclude that interactive-syntax is more readable than pure text if
What value of `==ANSWER==` is required to make this test pass?

A

```
[[:- :- :x :x :-]
 [:- :- :x :- :-]
 [:- :- :- :- :x]
 [:x :- :- :- :-]
] 1 :down-right
==ANSWER===)
```

B

```
[[:- :- :x :x :-]
 [:- :- :x :- :-]
 [:- :- :- 1 :x]
 [:x :- :- :- :-]
] 1 :down-right
==ANSWER===)
```

C

```
[[:- :- :x :x :-]
 [:- :- :x :- :-]
 [:- :- :- :- :x]
 [:x :- :- :- :-]
] 1 :down-right
==ANSWER===)
```

D Not Listed/Not Possible

Not Listed/Not Possible

Figure 16: Sample Multiple Choice Question
Given the following starter code, implement the logic for the display.

```clojure
(ns display.use
  (:require [display.core :include-macros true]
            [cljs.test :refer [is :include-macros ...]]))
(is (= (disp 0) (js/parseInt "1111110" 2)))
(is (= (disp 1) (js/parseInt "0110000" 2)))
(is (= (disp 2) (js/parseInt "1101101" 2)))
(is (= (disp 3) (js/parseInt "1111001" 2)))
(is (= (disp 4) (js/parseInt "0110011" 2)))
(is (= (disp 5) (js/parseInt "1011011" 2)))
(is (= (disp 6) (js/parseInt "1011111" 2)))
(is (= (disp 7) (js/parseInt "1100000" 2)))

Figure 17: Implementation of 7-Segment Display
• How long have you been working with Clojure/ClojureScript?
• How long have you been working with JavaScript?
• Roughly what size is the largest Clojure/ClojureScript codebase you’ve worked on?
• Roughly what size is the largest JavaScript codebase you’ve worked on?
• How would you rate your proficiency with Clojure(Script)?
• How would you rate your proficiency with JavaScript?
• How was your experience using VISr during this survey?
• If available in your preferred languages and text editors, would you use interactive-syntax in your own projects?
• Where would you use interactive-syntax?
• What roadblocks are stopping you from using interactive-syntax in your projects?
• Do you have any other feedback?

Figure 18: Exit Questions

more participants in the experimental group than the control group got the expected answers. The free-response questions have multiple correct answers. I performed a combination of manual and automatic checking to determine which responses were correct and which ones were incorrect.

12.2 RESULTS

Eighty-two participants took part in this study from January 1, 2022 to Feb 28, 2022. The majority of these participants opened the study, agreed to the initial consent, and answered no questions. This leaves roughly 30 participants who answered any questions, 17 of which answered multiple questions. Of these, 16 participants answered free response questions as well as multiple choice ones. Figure 19 gives the complete breakdown of the questions and variants that participants answered, with hexgrid questions on the left and 7-segment display questions on the right. The survey provided no purely textual variants for the free-response questions, therefore all responses were only for the visual variants.

Figure 20 shows the results for the multiple choice questions (RQ1). The top plot shows the number of respondents for each question variant, as well how many got the expected answer. In all questions
Figure 19: Question Response Rates
except disp-2, the visual variants got more responses than their textual counterparts.

The bottom half of figure 20 normalizes the results based on the number of responses each question variant received. The visual variant has a higher correct rate in hex-1, hex-2, hex-3, and disp-3. In contrast, the textual variant is higher in hex-4, disp-1, and disp-2.

To address RQ2, figure 21 gives the response rates for the two free form questions (left), as well as the normalized correct percentages (right). 16 participants attempted the hex question, 9 of which got the correct answer. The display question has 10 attempts, and 5 correct responses.

Eleven participants provided further feedback. Overall 10 people indicated that they would use interactive syntax in their code if it was available in their preferred language and toolset, while 1 person said they wouldn’t. Half of the participants had 3 or more years of experience in JavaScript or Clojure, the other half had less than 3 years of experience. Most participants have worked on codebases that involved 1,000 to 10,000 lines of Javascript or Clojure.

12.3 Discussion

One anomaly worth noting is hexagon question 3. Despite having both textual and visual variants, no participants responded to the textual variant. In contrast, the visual variant had a normal response rate. This strongly suggests that participants found the visual version easy to understand and its textual counterpart impossible or at least not worth attempting. The question is shown in figure 22 with an expected answer of #{:up :up-left :down-left}.

Another anomaly involves the inversion of responses for the second display question. Participants given the visual variant for the second to last question got the textual variant in the previous question, which may have caused them to drop out of the survey. This also fits because the textual variant of the first question required participants to calculate bits mentally, while the visual variant could directly see the translation of bits to display segments.

Continuing on with the 7-segment display, a first glance appears to show that visualization hurt understanding. However, this is only an artifact of the normalization. For example, question 1 in the 7-segment display had the same total amount of correct responses for both visual and textual, with additional people attempting the visual variant.

Overall, these responses seem to validate RQ1, indicating that programmers can read and understand programs that use interactive-syntax extensions. The responses for RQ2 are more mixed. They indicate that using interactive syntax did not hamper their ability to write code. It does not, however, demonstrate that interactive syntax was better or worse than a textual syntax for writing code. Additional user
Figure 20: Response Correctness for Multiple Choice Questions
Question

Figure 21: Response Correctness for Free Form Questions

What is the result of the following call to valid-moves?

(valid-moves
[[2 := :x]
[:: 1 3]
[:: :x ::]]
1)

Figure 22: Hexagon Question 3
studies with proper control groups are required to demonstrate that claim. In contrast, the racket/visr prototype evaluated in chapter 5 did hamper a users ability to write code.

12.3.1 Participant Feedback

Qualitatively all participants responded with enthusiasm about interactive syntax. Even the one who would not choose to use it was still interested in its development. A few participants ran into minor bugs with the prototype, but no show stoppers appeared. Some participants admitted to getting a pen and paper to determine the answers to the textual questions.

12.3.2 Threats to Validity

There are three main threats to this experiment’s validity. First, the sample size of participants was small. Second, participation in the study was self-selecting. Finally, despite having submission timestamps, it is impossible to know how long participants spent on each question.

While 82 participants started the survey, only 30 gave us any data whatsoever. When split between visual and textual variants, that means only 15 participants were answering each question variant. Also taking into account many of the questions were multiple choice, almost every expected answer had single digit responses. At such small numbers, it is hard to get any result beyond the base readability posed in RQ1.

In addition to a small sample size, the population was entirely self-selecting to people interested in visual and interactive syntax. Participants had getting an early view of core.visr as the only motivation for participating in the study. Fortunately there was a diverse set of participant experiences in those who did participate.

While not required for RQ1 or RQ2, this study does not take into account the length of time participants spent on each question. This means that while participants are able to read and write code with interactive-syntax extensions, it is impossible to say if it saves time using them. Anecdotally one could say that they can read code much faster using interactive-syntax extensions. But an additional study is required to prove it.
The evaluation from the preceding chapter covers the readability and writability aspects of interactive-syntax extensions (RQ1 and RQ2), but it does not cover the extensibility of the prototype (RQ3). A proper evaluation requires a participant to create extensions and use these creations in code. This evaluation should also allow participants to use other participants’ creations in code as well.

Doing this type of extended study in a controlled environment is infeasible. Another approach is to present core.visr publicly, and ask users to share any creations they find interesting. While this lacks the rigor of a controlled study, it can still serve as a semi-oracle. That is, if users create interesting things using core.visr, then at a minimum the language must be usable.

This chapter covers two examples of interactive-syntax extensions provided by Cameron Moy. The first example is a finite-state machine editor. The second example is an editor for plotting and visualizing data. Moy had access to a brief tutorial that covered the basics of interactive syntax, core.visr, and elIDE. I was available to answer occasional questions over email. In other words, Moy’s work was entirely self directed.

13.1 Finite State Machines

The first example is an interactive-syntax extension of a finite state machine, shown in figure 23. The automaton, my-aut, is a function defined on lines 1–4. This function takes a sequence of characters, and returns a boolean indicating if the sequence is in the language. Buttons exist to create new states, set states as accepting, and create edges between states. Additionally, nodes can be dragged and the entire display can be both panned and zoomed. The automaton in this example only accepts one sequence: “z”.

Two uses of my-aut are on lines 31–32. Function application semantics are used for the automaton to interact with the rest of ClojureScript.

Figure 24 displays the implementation for this editor. Lines 2–9 are the run-time semantics, while line 10 is the edit-time semantics. The elaboration is a simple deterministic finite automaton acceptance algorithm that always starts at state 0. The rendering method calls out to flow-component, a custom automaton renderer made by Moy.
Figure 23: An finite state machine editor with tests

```clojure
def my-aut
  )👁 (λ)z
1
0
○
◎
(println (my-aut ["x"])) ; => false
(println (my-aut ["z"])) ; => true
```

Figure 24: Implementation of Finite State Machine Editor

This renderer also makes use of an existing graph renderer for vanilla JavaScript, react-flow-renderer. The entire implementation is about 150 lines.

13.2 DATA VISUALIZATION

Moy’s second contribution was a small data visualization library, shown in figure 25. This library draws formatted ClojureScript data structures. The user can select the plot type and the color for that data. Selected options are stored with the visualization, so they can be archived with the code. These visualizations can additionally be exported to standard image formats such as SVG. Two data points,

---

2 https://reactflow.dev/
shown on lines 1–3, are stored in this example. Lines 5, 18, and 31 provide three visualizations of those two points.

Figure 26 shows the implementation for this editor. Note that line 2 shows the lack of an elaborator. As a result, the primary purpose of this editor is to allow scientists to visualize the plots themselves. When running the initial sample, the data on lines 1–3 of figure 25 prints to the console three times. Lines 4–17 provides a small set of controls for the plot, while the actual visualization is handled on line 19. As before, the actual visualization is handled by a vanilla JavaScript library, this time react-vega. The entire implementation fits into 25 lines of code.

13.3 DISCUSSION

These two small case studies demonstrate that core.visr does satisfy the extensibility criteria. While the code samples provided here are small, they both link to full fledged JavaScript libraries for ordinary GUI programs. Furthermore, these were not libraries that I prepared for Moy or even recommended. Rather, Moy searched the internet for possible packages and linked them to core.visr.

With all three aspects demonstrated, I conclude that interactive syntax can be both useful and usable.

3 https://vega.github/
```clojure
(defvisr Vega
  (elaborate-fn [this] this)
  (render [this]
    [:<>
     [:select
      {:value (-> @this ... }
      [:option {:value "bar"} "Bar"]
      [:option {:value "line"} "Line"]
      [:option {:value "point"} "Point"]
     [:br]
     [:input {:type "color"
              :id "color-picker"
              :style {:height "14px"}
              :value (-> @this :spec :mark :color)
              :on-change #(reset! this (assoc-in @this [:spec :mark :color] {.. % -target -value}))}
     [:> VegaLite @this]]))
```

Figure 26: Implementation of Data Visualization Editor
Given that part I showed that racket/visr is useful, and part II showed that core.visr is usable, it is worth taking a moment to compare these two systems.

Figure 27 employs a table that compares the implementations of racket/visr and core.visr for interactive-syntax extensions. It compares how well each system’s design implements the goals and desiderata from chapter 3.

The core.visr prototype meets the ‘GUI reuse’ criteria, but fails the ‘low friction’ criteria. This is due to ClojureScript’s limitation requiring macro definition and uses to be in separate files. In practice, the ability to use a rendering engine with multiple decades of engineering offsets the relatively higher cost of splitting uses and definitions into separate files. Additionally, extensions that affect only run-time semantics, not compile-time semantics, can be placed in the same file.

The other area where core.visr falls short is in its sandboxing capabilities. While not a problem for all of the user studies reported in this documented, core.visr does highlight the need for future research on the interface between interactive-syntax and web-security. Fortunately the limited sandbox provided by Stopify and web browsers means interactive-syntax is, at worst, still as secure as visiting a web page.

Finally, because core.visr is based in ClojureScript, it inherits all of the problems and limitations of the ClojureScript macro system. Namely, editor elaborators are not hygienic. Additionally, since editor definitions and uses must be split into separate files, meta-editors must also be split into a separate file. Function editors reduce the severity of the first limitations, but cannot be used to construct meta-editors.
<table>
<thead>
<tr>
<th>Property</th>
<th>Racket</th>
<th>ClojureScript</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mixing Textual and Interactive-Syntax</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>IDE Agnostic</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>GUI Reuse</td>
<td>✗*</td>
<td>✓</td>
</tr>
<tr>
<td>Interactive Syntax is Syntax</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Persistent</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Compositional</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Low-Friction</td>
<td>✓</td>
<td>X**</td>
</tr>
<tr>
<td>Full Programming Language</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Sandbox for IDE</td>
<td>✓</td>
<td>X***</td>
</tr>
<tr>
<td>Sandbox for Code</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Macro Hygiene</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>Meta-Edit Time</td>
<td>✓</td>
<td>✓**</td>
</tr>
</tbody>
</table>

* Requires a shim to generate GUIs from a common source.
** ClojureScript macros must be placed in a separate file.
*** Stopify provides limited sandboxing for the IDE.

Figure 27: Comparison of racket/visr and core.visr
Part III

USEFULNESS AND USABILITY ACROSS LANGUAGES
The architecture presented in the preceding part brings a usable design for interactive-syntax extensions to languages whose GUI frameworks fit naturally with the DOM. While the prototype focuses on ClojureScript, a similar design works for other languages that naturally run in the DOM, such as JavaScript or TypeScript.

However, not all languages have support for running in the context of a DOM. These languages have three options to gain support for interactive-syntax extensions: (1) bring the DOM to their language, (2) bring their language to the DOM, or (3) create an interface that bridges the gap between their language and the DOM.

The first option, bringing the DOM to the language, is the easiest to understand but the hardest to accomplish. This option requires developers to implement the DOM, or a DOM-like system, in the language’s native environment. As described in part I, racket/visr attempts this approach by reimplementing a GUI that can be embedded in code editors.

Next, the second option, bringing the language to the DOM, is much easier to implement, but not always possible. Taking this path requires developers to write a compiler, or interpreter, from the host language into DOM code. Languages such as Clojure take this approach when developing ClojureScript. The core.visr library described in part II takes advantage of this approach.

Finally, the third option, bridging the gap between language and the DOM, is less straightforward than the other two. It promises to bring interactive syntax to languages that do not, or can not, either move to the DOM or reimplement a DOM system in their own environment. Bridging the gap entails building a small FFI layer to the DOM language to the corresponding target language.

The following chapters focus on the design of such a bridge for Racket. First, chapter 16 outlines the requirements of the design and provides a simple implementation using core.visr. Then, chapter 17 returns to the Tsuro example for Racket, but using the bridge to the DOM for interactive syntax. Finally, chapter 18 covers a case study of another developer who makes use of this bridge to create a visual variant of PLT Redex [Felleisen et al. 2009; Klein et al. 2012].
The architecture described in the preceding part can be altered to support other languages that can not execute immediately in the context of a DOM. This alternate architecture first splits the language into a front-end GUI library and a back-end language server, and then bridges the gap between them.

This chapter describes the altered architecture. First, it lays out the requirements for the front-end library and back-end server. Then it provides a basic implementation for Racket, with a ClojureScript based GUI library. The next chapter contains an example program using this architecture.

16.1 THE BRIDGE DESIGN

This alternate architecture splits the interactive-syntax into a front-end editor and a back-end language server. The language server can run either on the user’s machine or an independent server. The front end, however, is a webpage that must be executable directly in the DOM and capable of rendering React components. These two components will communicate using some mechanism, such as WebSockets, to exchange program text and React objects. Figure 28 provides a diagram of this architecture.

The front end is responsible for the program source. It provides a text editor for program creation and manipulation, as well as the context for running interactive-syntax extensions. Event handlers for these extensions become part of the React component. When the state for an editor changes, the front-end is responsible for ensuring that the program source updates accordingly. The best way for an editor to send a message to the language server is through an updated state. This state must be stored in an encoding, such as a JSON object, that both the front-end and language server can understand.

The language server is responsible for actually running the program’s run-time code. Each interactive-syntax extension must generate the code for a state, view, and event handler inside a React component. These components are sent to the front end through some foreign function interface. React components will be sent as strings, which are evaluated in a sandboxed browser context. The semantics for what is interactive syntax rather than traditional syntax is language specific. Therefore, the language server sends each editor with an accompanying source location and range. When the front end renders the

1 https://websockets.spec.whatwg.org/
program for the user, it replaces this range with the provided extension. Additionally, the language server must also provide the result of running the program back to the end user.

This design provides a truly language-agnostic environment for editing programs with interactive-syntax extensions. Many languages contain direct, but limited, support for embedding JavaScript in their code. A language can support JavaScript directly, such as ClojureScript, or it can support it via a third party DSL like Urlang\(^2\) for Racket, WebAssembly for Rust, etc. This design, however, is only partially editor agnostic. The front end and back end must coordinate to use a shared web-based GUI language, e.g., React.

16.2 THE BRIDGE FOR RACKET

Pairing up ClojureScript and Racket makes the interaction between the front and back ends is straightforward. Both languages are heavily S-Expression based, and while the semantics of both languages differ wildly, they do both share similar data structures. Due to the similarity of these data structures, S-Expressions can be used as the common medium to transfer editor state between language server and front-end. Additionally, source locations can be sent using ClojureScript’s metadata parser and Racket’s reader macro extensions. These two mechanisms allow S-Expressions to be tagged as interactive-syntax extensions.

Since only data is shared, developers must separate the code into strictly run-time or strictly edit-time code. The upshot of this separation is that a developer can easily edit code using interactive syntax, and then run that code using the Racket interpreter. Unfortunately,

\(^2\) https://github.com/soegaard/urlang
the downside is that the implementation of these editors must remain separate from Racket.

To make this approach work, core.visr for Racket requires three components. First, the reader must be extended to accommodate S-Expression tagging. Second, a small FFI layer must be introduced to bind React code to a Racket object. Finally, a new variant of the define-interactive-syntax form must be created to that uses core.visr instead of racket/visr.

16.2.1 The Reader

The reader extension mechanism follows the same pattern described in chapter 6. The first line of any program using core.visr for Racket must use the meta-language visr-support, which has two effects on the program. First, it introduces a special reader form: `{editor}, which tags an S-Expression, but is otherwise ignored by the main reader. Second, it adds support for the syntax of ClojureScript keywords, sets, dicts, and vectors.

Figure 29 shows a simple example of a program using this meta-language. Line 1 shows this is a Racket program, but makes use of interactive-syntax extensions. Line 5 shows the textual representation of a small Counter extension. The tag indicates that the following S-Expression is intended as interactive syntax, as well as which editor to use. Finally, Counter+elaborate is a Racket macro introduced by define-interactive-syntax. Its implementation is elided (line 3) in this example.

16.2.2 The FFI

The FFI is what allows running React components, powered by ClojureScript, in the syntactic context of a Racket program. This is not a traditional FFI and does not support any run-time language interoperability. Rather, it statically binds a Racket identifier to a React object. The IDE can use this binding information, in conjunction with the tagging information from above, to turn S-Expressions into interactive editors. In principle a run-time library could also use this binding to
serve as the GUI for the program, but this document does not cover this process.

The syntax for this FFI is:

\[
\text{(define-react-component component-id (file-location function))}
\]

where \text{component-id} is the Racket identifier used for graphical components, and \text{file-location/function} are used by the IDE to find the React component. The \text{Counter} example from figure 29 looks as follows:

\[
\text{(define-react-component counter-render (counter.core react-counter))}
\]

This defines a new component, \text{counter-render}, whose implementation can be found in the ClojureScript file \text{counter.core}, and the function \text{react-counter}.

16.2.3 \textit{The Semantics}

Finally, a \text{define-interactive-syntax} form ties these React components with tagged expressions described above. Unlike the version of \text{define-interactive-syntax} presented in chapter 4, this form is not class based. Instead, it inherits its design from \text{defvisr} as described in chapter 9.

The definition for the counter extension is:

\[
\text{(define-interactive-syntax Counter (render counter-renderer) (elaborate this ...elided...))}
\]

where \text{counter-renderer} is the React component from above. The \text{elaborate} block is standard macro. It is similar to \text{define-elaborate} from chapter 4. The next chapter gives an example of an elaborator.
This cross language architecture can also build on the Tsuro example from chapter 2 and chapter 10. In fact, the entire implementation for the tile editor can fit into less than 20 lines of code, shown in figure 30. This tile editor uses the compile-time semantics from the Racket Tsuro implementation, and the edit-time semantics from the ClojureScript Tsuro implementation.

The first line of this example shows that the language is Racket, with the extended reader described in the previous chapter. Lines 3–5 contain the import statements. The react-cross/cljs module (line 3) provides the bindings for define-react-component as well as define-interactive-syntax used later. The "tsuro.rkt" file contains the tsuro% class that is used in chapter 2. Finally, line 7 imports the visual react component displayed at edit time. This is the same editor defines on lines 7–14 of figure 15 back in chapter 10.

With these imports in place, lines 9–11 contain the implementation for the extension itself. The renderer (line 10) is a plain React component, while the elaborator (line 11) is a traditional Racket macro. For both the renderer and elaborator, the editor’s state is passed as a single variable, this. The state is a dictionary which maps the key 'edges for Racket and :edges for ClojureScript to a set containing each connection in the tile. Using a single implicit state follows the design of core.visr over racket/visr.

Finally, lines 13 and 18 show an example of this Tsuro tile used in a unit test. Since the visual tile elaborates to an object, this unit test calls the rotate method on the tile object. Rather than mutating the tile, this method returns a new tile object that is rotated by the specified amount, in this case by 90 degrees.

One major advantage of this bridge approach is that the elaborator for the tile$ extension gets the hygienic capabilities and explicit phase separation of the Racket language. The elaborator, for example, runs code explicitly at compile time, which produces code that runs at run time. The syntax symbol (#') indicates that the following expression is run time code, while the unsyntax symbol (#:,), indicates that the following expression is compile time code that evaluates to a syntax object. For more details on how this macro system works see chapter 4.

The main downside of this approach is that either the GUI for the Tsuro editor still has a separate implementation from the rest of the application GUI, or the GUI portion of the application must be written in ClojureScript. Fortunately, the relative maturity of DOM-based environment dramatically offsets this cost. Additionally, this
#lang visr-support racket

(require react-cross/cljs
         rackunit
         "tsuro.rkt")

(define-react-component tile-component (tsuro.core tile-view))

(define-interactive-syntax tile$
    (render tile-component)
    (elaborate this #`(new tile% [edges #,(dict-ref this 'edges)]))))

(check-equal? (send rotate 90) 👁
Add Edge
(λ)
              )👁 
Add Edge
(λ)

Figure 30: Racket Tsuro Using core.visr
limitation will evaporate when Racket eventually gets a true DOM-based variant.
The bridge architecture presented here allows interactive-syntax extensions to work in other languages. The question remains, however, whether interactive syntax remain usable even when making use of this Frankenstein’s monster [Shelley 1818]? The answer is yes, interactive syntax does remain usable even when using the cross-language approach presented here.

The readability and writability criteria are unaffected by this cross-language approach, the edit-time code for each extension remains in the ClojureScript. Indeed, only the extensibility criteria is affected by this bridge architecture. Previously, the way I demonstrated usability was through case studies created by developers who have not worked on core.visr in any way. This chapter is no different; it describes one noteworthy case involving Cameron Moy and PLT Redex [Felleisen et al. 2009; Klein et al. 2012].

PLT Redex is a Racket-based embedded domain-specific language for describing programming language semantics. A program written with PLT Redex attempts to look like the semantics in an actual paper. Unfortunately, the text typed into a computer cannot look exactly like the math used in papers. To fix this, Cameron Moy set out to enrich PLT Redex with interactive syntax.

### 18.1 Runnable Semantics

Figure 31 shows the entire implementation for a fully runnable Redex model of the Simply Typed Lambda Calculus [Church 1940]. The syntax for the language is shown on lines 8–31. Lines 33—56 show the typing rules for the language. Finally, lines 58–60 give the reduction relationship for \( \beta \)-reduction.

The following is an example program in this calculus—the identity function:

```
(define id-eg (\(x:\text{Int}\).x))
```

Standard Redex functions, like `redex-match` work on this program as expected:

```
> (redex-match? lc v id-eg)
#t
```

The program is runnable just as if it was written in the plain textual variant of Redex:
Figure 31: Simply Typed Lambda Calculus Made Using core.visr
> (eq? (apply-reduction-relation
   lc-reduce
   (term ,id-eg ,id-eg))
   id-eg)

#t

Note that the result of applying this reduction relation is itself a piece of visual syntax. In other words, by using core.visr for Racket, Moy created a fully visual lambda calculus.

18.2 IMPLEMENTATION

The implementation of the visual semantics extension is less than 550 lines of code. The visual and edit-time code consists of roughly 250 lines of ClojureScript code. The Racket code is slightly larger at 300 lines, with 100 lines dedicated to interoperation with ClojureScript. Finally, as shown in figure 31 the model itself fits in roughly 60 lines of code.

From a qualitative perspective, Moy provided highly positive feedback about his experience with both core.visr and elIDE. However, he did indicate that this cross-language approach does demand that programmers be versed in two different languages and ecosystems.
Part IV

**VIDEO: AN EXTENDED CASE STUDY**

This part of the dissertation is derived from collaborative work with Chang and Felleisen. [Andersen et al. 2017]
Imagine preparing video recordings for all of the talks for at a conference. After recording the clips, the feed of the presenter must be combined with the presenter’s screen, the sound feed for the speaker, and yet another one for audience questions. Also, a start sequence, an end sequence, and various watermarks should be added to the video.

Once one video is put together, the same process must be repeated for the next conference talk and the next and so on. Worse, even though some editing steps involve creativity, the process becomes so monotonous that it reduces the creative spirit for when it is truly needed.

The problem cries out for a functional domain-specific language, especially because the state of the art for video editing suggests that professionals in this domain already think “functionally.” To wit, professionals speak of “non-linear video editing” (NLVE) to highlight the idea that the process is non-destructive. Technically, the editing process separates a descriptive phase—what the eventual video is supposed to look like, given existing tracks—from the rendering phase—which actually creates the video clip from these descriptions.

This part of my dissertation presents an extended case study of Video, a scripting language with interactive-syntax extensions for video production. Video turns video editing upside down. Instead of sitting for hours on end in front of some traditional NLVE GUI, a professional can now spend a few minutes in front of an IDE to create a Video script, which creates a series of Video clips. A Video script is just a sequence of expressions, which describes fragments of a video clip, and definitions, which introduce constants and functions for video clips. Running such a script turns this description into suitable “assembly code” for a video renderer. Additionally, core.visr combined with Racket provides a perfect opportunity for users to retain the creative elements of a graphical NLVE, while also gaining the programmatic power of a programmable programming language, and interactive syntax for expressing it all.

To start, chapter 20 lays out the design of the Video language. Next, chapter 21 shows how to use core.visr to add interactive NLVE editors to Video. Finally, chapter 22 lays out how to implement the language.
A non-linear video editing distinctly separates the description of a video clip from the rendering action on it. Specifically, an editor (as in the tool) needs a description of what the final video should look like in terms of the given pieces. The action of rendering this video is a distinct second step. Going from this assessment to a language design requires one more idea: abstraction. For example, a description of a video composition should be able to use a sequence comprehension to apply a watermark to all images. Similarly, a professional may wish to create one module per ICFP presentation in order to make up a complete ICFP channel in a modular fashion. And of course, the language must allow the definition and use of functions because it is the most common form of abstraction.

The Video language gets to the heart of the domain. Each Video program is a complete module that intermingles descriptions of video clips and auxiliary definitions. It denotes a Racket module that exports a playlist description of the complete video. One way to use a Video module is to create a video with a renderer. A different way is to import it into a second Video module and to incorporate its exported playlist into another video.

Figure 32 displays a simple Video script. It consists of five expressions, each describing a piece of a video clip. Right below the third part of the video description, it also contains two definitions, which introduce one name each so that the preceding multitrack description does not become too deeply nested. When the renderer turns this script into an actual video, it turns the five pieces into sequence of images, taking into account the transitions between the first and second fragment and the fourth and the fifth.

In essence, the script in figure 32 assembles the visual part of a simple conference video. What is missing is the audio part. Naturally, a Video programmer should abstract over this sequence of expressions, plus the audio processing, and create a suitable library. Figure 33 shows what the same script roughly looks like after a Video programmer has encapsulated an abstraction over the script in figure 32 as a utility library. This program uses the imported conference-talk function to combine a recording of the speaker, a capture of the slides, and the audio. As mentioned, line 1 of the program specifies that this module is written in the Video language. Line 3 imports the library that defines the conference-talk function. Line 5 produces the video that this module describes. Finally, the remainder is a sequence of definitions that introduce auxiliary constants.
01 #lang video
02
03 (image "splash.png" #:length 100)
04
05 (fade-transition #:length 50)
06
07 (multitrack (blank #f)
08    (composite-transition 0 0 1/4 1/4)
09    slides
10    (composite-transition 1/4 0 3/4 1)
11    presentation
12    (composite-transition 0 1/4 1/4 3/4)
13    (image "logo.png"
14     #:length (producer-length talk)))
15
16 ; where
17 (define slides
18    (clip "slides05.MTS" #:start 2900 #:end 80000))
19
20 (define presentation
21    (playlist (clip "vid01.mp4")
22      (clip "vid02.mp4"
23       #:start 3900 #:end 36850))
24
25 (fade-transition #:length 50)
26
27 (image "splash.png" #:length 100)

Figure 32: A First Video Script
#lang video
(require "conference-lib.vid")
(conference-talk video slides audio 125)
; where
(define slides (clip "slides05.MTS" #:start 2900 #:end 80000))
(define video (playlist (clip "vid01.mp4") (clip "vid02.mp4")
  #:start 3900 #:end 36850))
(define audio (playlist (clip "capture01.wav") (clip "capture02.wav")))

Figure 33: A Video Description of a Conference Talk

Figure 34 shows the essence of the utility library, also written as a Video module. Explaining its construction introduces enough of Video’s primitives and combinators to get a sense of what the rest of the language looks like. First I explain Video’s primary linguistic mechanisms, modules and functions. Then, I describe basic producers: images, clips, colors and so on, following up with a discussion of the basics of how to combine these producers into playlists and multitracks. To make compelling examples, I introduce transitions, filters, and properties. Finally, I describe the interface for displaying videos and rendering Video programs.

20.1 ESSENTIAL VIDEO

Video modules consist of a series of interleaved expressions, definitions, and import/export forms; functions have the same shape as modules but without the import/export forms. Video enforces different scoping rules, and assigns slightly different meaning to these constructs, than Racket does. In both cases, definitions are valid in the entire scope—that is, the entire module or the entire function body. The expressions must describe video playlists. Modules and functions differ in that the former provides a video, while the latter returns one. Furthermore, Video modules are first-order entities that can be compiled separately, while functions are actually first-class values.

Now, take a second look at figure 34. Lines 5 through 7 show the function header. The rest of the code describes the function body (lines 7–28). Functions in Video are declarative; in particular, line 8 is the producer returned by this function. The remaining subsections explain the Video language in sufficient detail to understand the rest of this
(provide conference-talk)

; Describes an edited conference video with appropriate feeds
; Producer Producer Producer Positive-Integer -> Producer
(define (conference-talk video slides audio offset)
  (multitrack clean-video clean-audio)
  ; where
  (define clean-audio
    (playlist (blank offset)
      (attach-filter audio
        (envelope-filter 50 #:direction 'in)
        (envelope-filter 50 #:direction 'out)))
    (define spliced-video
      (multitrack (blank #f)
        (composite-transition 0 0 1/4 1/4
          slides
        (composite-transition 1/4 0 3/4 1
          video
        (composite-transition 0 1/4 1/4 3/4
          (image "logo.png" #:length (producer-length talk)))
      (define clean-video
        (playlist (image "splash.png" #:length 100)
          (fade-transition #:length 50
            spliced-video
          (fade-transition #:length 50
            (image "splash.png" #:length 100))));

Figure 34: A Video Function Definition
code. Specifically, I explain individual features of the language and how they improve the video editing process.

20.2 PRODUCERS

The producer is the most basic building block for a Video program. A producer evaluates to a data structure that denotes some sort of multimedia object: audio clips, video clips, pictures, and so on. Combinations of producers are themselves producers, and they can be further combined into yet more complex producers.

The simplest type of producer, clip, incorporates traditional video files. The clip producer converts the file into a sequence of frames. Developers use clip to import recordings from files, such as a conference talk, into scripts:

```
#lang video
(clip "talk00.mp4")
```

Unlike clip, image creates an infinite stream of frames. Video’s combination forms truncate these streams to fit the length of other producers. Additionally, developers can use the #:length keyword when they want a specific number of frames:

```
#lang video
(image "splash.png" #:length 2)
```

20.3 PLAYLISTS

A video is usually a composition of many producers. Video provides two main ways for combining them: playlists and multitracks. Roughly speaking, playlists play clips in sequence, while multitracks play clips in parallel. Any producer can be put in a playlist, including another playlist. Each clip in the playlist plays in succession. Frequently, video cameras split recordings into multiple files. With playlist, developers can easily stitch these files together to form a single producer:
Developers cut playlists and other producers to desired lengths with the #:start and #:end keywords. This capability is included because video recordings frequently start before a talk begins and end after the talk finishes; see figure 35a. Recall that while define is located below the video description, its scope includes the preceding expressions.

20.4 TRANSITIONS

Jumping from one producer in a playlist to another can be rather jarring. Scripts can reduce this effect with transitions: fading, swiping, etc. These transitions merge the two adjacent clips in a playlist and are placed directly inside of (possibly implicit) playlists. Expanding on the example from figure 35a, fade transitions are used to smooth the transition from logo to the talk. Figure 35b illustrates this point.

Every transition in a playlist actually shortens the length of its frame sequence, because transitions produce one clip for every two clips they consume. Additionally, a playlist may contain multiple transitions. Such a playlist still specifies a unique behavior because playlist transitions are associative operations. Thus, multiple tran-
Multitracks play producers in parallel. Like playlists, they employ transitions to composite their producers. Syntactically, `multitrack` is similar to `playlist`. The `multitrack` form consists of a sequence of producers and creates a new `multitrack` producer. Again, transitions are included within the sequence to combine tracks; see figure 35c. This example uses `composite-transition`, which places one producer on top of the other. The four constants specify the coordinates of the top-left corner of the producer and the screen space that the top producer takes. Here, the producer following the transition appears in the top-left hand of the screen and takes up one quarter of the width and height of the screen.

Transitions within a `multitrack` are not associative; instead, the `multitrack` form interprets transitions in left to right order. Videos that require a different order can embed a `multitrack` inside of another one, because a `multitrack` is itself a producer. Using multiple transitions allow producers to appear side by side rather than just on top of each other. Expanding on the running example in figure 35c, figure 35d describes a conference video where the recording of the presenter goes in the top left while the slides go on the right. This example adds a `composite-transition` to the previous example and places the slides and the recording over a single blank producer. Blank producers are empty slides that act as either a background for a `multitrack` or a
Figure 35 (continued): (c). Example Using Multitracks

Figure 35 (continued): (d). Example Using Inlined Transitions in Multitracks
filler for a playlist. In this case, the blank producer is providing a background that the slides and camera feeds appear on.

20.6 Filters

Filters are similar to transitions, but they modify the behavior of only one producer. In other words, filters are functions from producers to producers. Among other effects, filters can remove the color from a clip or change a producer’s aspect ratio. Conference recordings frequently capture audio and video on separate tracks. Before splicing the tracks together, a developer may add an envelope filter to provide a fade effect for audio.

A script may use function application notation to apply filters or attach them to producers with the #:filters keyword. Figure 35e shows an example of a filter being attached to an audio track that is itself composited with the video of the composited talk in figure 35d.

20.7 Properties and Dependent Clips

Producers use two types of properties to store information: implicit and explicit properties. Implicit properties are innate to clips, for example, length and dimensions. Explicit properties must be added by the program itself.

The properties API comes with two functions:
(multitrack
  (blank #f)
  (composite-transition 0 0 1/4 1/4)
  (clip "slides.mp4")
  (composite-transition 1/4 0 3/4 1)
  talk
  (composite-transition 0 1/4 1/4 3/4)
  (image "logo.png"
    #:length (get-property talk 'length)))

; where
(define talk ⟨⋯⟩
  ⟨⋯⟩)

Figure 35 (continued): (f). Example of Adding a Watermark

- (set-property producer key value) creates an explicit property. It returns a new producer with key associated with value.
- (get-property producer key) returns the value associated with key. If the property is set both implicitly and explicitly, the explicit property has priority.

Explicit properties provide a protocol for communicating information from one clip to another. Implicit properties exist for the same purpose except that they store information that is implicitly associated with a producer, such as its length. For example, a conference video may have to come with a watermark that is the same length as the captured conference talk. A script that performs this operation can be found in figure 35f.

20.8 FROM PROGRAMS TO VIDEOS

A Video module may be incorporated into a program or understood as a stand-alone program. In the first case, another Video or Racket module may require the Video module and incorporate its export into its own code. In the second case, a user can hand the Video script to a renderer that either plays the video on a screen or saves it to a file.
By default, a Video module implicitly provides a producer. Any module that wants to use this implicitly created video imports it with the external-video form. For an example, consider this two-line module and what it denotes:

```racket
#lang video
(image "splash.png" #:length 100)
(external-video "talk.vid")
```

The module’s first line sets up a splash screen, the second line incorporates the external Video module.

A renderer converts Video scripts to traditional videos. Having a dedicated rendering pass allows users to set various visual properties such as aspect ratio, frame rate, and even output format separately. The simplest renderers, dubbed render and preview, are functions that consume a producer and display it in a separate window. At DrRacket’s REPL, developers can apply this function directly (left half of figure 36). While render just displays the video, preview adds playback controls, and even gives developers the ability to preview a video excerpt. Another renderer, called preview-video is a function that consumes a path to a Video script and plays it in a newly opened...
window (right half of figure 36). This functionality is available outside of the IDE so that non-programmers may view the videos, too. However, programmers editing Video programs from within an IDE can use an interactive-syntax extension to preview the Video they are editing, as discussed in the next chapter.
Adding Interactive Syntax to Video

Some videos are best expressed with a graphical NLVE. Video therefore comes with core.visr support for embedded NLVE widgets. Unlike other NLVEs with scriptable APIs, the NLVE widget is actually part of the Video language. A developer may place an NLVE directly into a script. See figure 37 for a sample of such nesting.

Consider the actual scenario when a hardware failure during a talk prevented the capture of the speaker’s screen. Fortunately, the speaker supplied a copy of their slide deck as a PDF document. While the captured video can still be recreated by using the slide deck, a decision has to be made concerning the duration of each slide. If a plain-text Video script were to use this method, it would inevitably contain a bunch of “magic numbers.” Embedding NLVE widgets into the code explains these “magic numbers” to any future reader of the code and is thus a cleaner way to solve the problem. Figure 38 illustrates this point with a simplistic example, with a module of mostly magic numbers, while figure 39 shows how an embedded NLVE explains the numbers directly. In both cases a developer must manually determine the screen time allocated to each slide. However, using the widget gives the author a graphical representation of the layout, thus speeding up development time. Additionally, future authors can more easily tweak the times by dragging and resizing clips in the widget.

Graphical NLVEs are producers and in turn first-class objects in Video. They can be bound to a variable, put in a playlist, supplied to a multitrack, and so on. Integrating the graphical and textual program in this manner allows users to edit videos in the style that is relevant for the task at hand. For example, the program in figure 37 shows an implementation of the conference-talk function from chapter 20, now implemented using NLVE widgets with embedded code snippets.

Video relies on the Racket ecosystem to get REPL-style feedback needed for quick video editing. As described in chapter 20 the preview function shows a low-resolution (but real time) preview of the video being edited. Interactive-syntax extensions make it possible to view this preview right in code, as figure 40 shows. It renders the preview for a producer right in code, and even offers simple graphical editing capabilities for that preview.

This preview editor serves three primary functions. First and foremost, it finds video files in the user’s filesystem and is able to display those files. Second, it can partially evaluate a Video program to produce the intermediate video. Finally, it offers some limited capabilities to edit the video graphically, right within the preview window.
Figure 37: Mingling Graphical NLVE Widgets Inside of Video Scripts
#lang video

(apply playlist
(for/list ([slide (directory-list slides)]
[time (in-list slide-times)])
(image slide #:length (* 48 time))))

; where
(define slide-times
(list 75 100 105 120 50 30 30 19 3
10 50 15 33 250 42 20 65
13 9 25 37 25 13 30 39 45))

---

**Figure 38**: Slide Reconstruction Using Magic Numbers

---

**Figure 39**: Slide Reconstruction Using a NLVE Widget

---

**Figure 40**: Simple Extension for Previewing and Editing
The partial evaluation done by the preview editor does have two non-trivial requirements. First it necessitates evaluating some run-time code during edit time. Racket’s phase system is not set up to do this, so a second Racket VM does this partial evaluation. Second, it also requires a version of the preview renderer to run in the background as the user writes code. This preview renderer runs in tandem with the second Racket VM.
IMPLEMENTING THE VIDEO LANGUAGE

Using the Racket ecosystem allows developers to implement languages quickly and easily. Furthermore, these languages compose so that modules written in one language can easily interact with modules in another. Best of all, the implementation of a language may take advantage of other language technologies, too. The upshot here is that implementing Video is as simple as implementing video-specific pieces, while leaving the general-purpose bits to Racket.

Video’s implementation makes heavy use of the Racket ecosystem. It consists of three major components and accounts for approximately 2,400 lines of code: a surface syntax, a run-time library, and a rendering layer. Of the code, about 90 lines are specific to the syntax and 350 lines define the video-specific primitives the language uses. The remaining lines are for the FFI and renderer. The video-specific primitives serve as adapters to imperative actions that work on Video’s core data-types; they are implemented using standard functional programming techniques.

This section explains how a developer can implement a DSL in Racket (section 22.1), with Video serving as the running example (section 22.2). Not only is Video a Racket DSL, but part of the implementation of Video is implemented using additional DSLs created specifically for implementing Video (section 22.3).

22.1 CREATING LANG UAGES, THE RACKET WAY

Creating Racket DSLs means removing unwanted features from some base language, adding new ones, and altering existing features to match the semantics of the new domain.

Adding and removing features is simple, because a language implement- ation is a module like any other module in Racket. Removing a feature is simply a matter of not re-providing it from the host language. See figure 41 for an example. Line 4 uses the except-out keyword to remove the definition of set! from racket/base, while all-from-out re-exports all remaining features from racket/base.

In addition to these operations, adding new features is simply a matter of defining the new features and exporting them. Developers do so in the same manner as a programmer who augments the functionality of a library via a wrapper module.

In contrast, modifying existing features requires slightly more work. Specifically the module must define a syntax transformation in terms of the old language and rename this transformation on export.
01 #lang racket/base
02
03 (provide (rename-out [boo:set! set!])
04     (except-out (all-from-out racket/base) set!))
05
06 (define-syntax (boo:set! stx)
07   (syntax-parse stx
08     [(_ id val)
09      #'(begin (log-warning "Boo!!! Reassigning ~a" id)
10       (set! id val))])))

Figure 41: Logging Assignment Statements to Expose Ill-behaved Functional Programmers

Let us illustrate this idea with a simple example. Many functional programmers do not like assignment statements; at Clojure conferences, programmers who admit to their use (via Java) are booed on stage. So, imagine creating a language like Racket that logs a boowarning of any use of set!, Racket’s variable assignment form. The set! provided by racket/base provides the functionality for assignment, while log-warning from the same language provides the logging functionality. All I have to do is define a new syntax transformer, say boo:set!, that logs its use and performs the assignment. From there, I need to rename boo:set! to set! in the provide specification. This renaming makes the revised set! functionality available to programmers who use the new and improved Functions-first variant of Racket. Figure 41 displays the complete solution.

Now recall that Racket’s syntax system supports several interposition points that facilitate language creation: #%app for function application, #%module-begin for module boundaries, #%datum for literals, and so on. The purpose of these points is to allow language developers to alter the semantics of the relevant features without changing the surface syntax.

Figure 42 displays a small, illustrative example. Here, a language developer uses a strict #%app to construct a lazy form of function application. The #%app protocol works because the Racket compiler places the marker in front of every function application. Thus, language developers only need to implement their version of #%app in terms of an existing one.¹

¹ My advisor witnessed this booing at a recent Clojure conference, which has the largest, most successful community of commercial functional programmers.

² Indeed, the Racket family of languages comes with the lazy language [Barzilay and Clements 2005], which uses exactly this interposition point to convert racket into an otherwise equivalent language with lazy semantics.
When a programmer uses this new language, the Racket syntax elaborator inserts #%app into all regular function applications. The elaborator resolves this reference to the imported version, written as #%app_lazy. From there, Racket redirects to #%lazy-app, which expands into #%app_base, Racket’s actual application. Here is what the complete process looks like:

\[
(f \ a \ b \ c \ \ldots)
\]

elaborates to \(#%app_lazy f \ a \ b \ c \ \ldots\)

elaborates to \(#%lazy-app f \ a \ b \ c \ \ldots\)

elaborates to \(#%app_base (force f)\)

\quad (\text{lazy } a) \ (\text{lazy } b) \ (\text{lazy } c) \ \ldots\)

The curious reader may wish to step through the elaboration via DrRacket’s syntax debugger [Culpepper and Felleisen 2010].

22.2 THE ESSENCE OF VIDEO’S SYNTAX

The implementation of Video’s syntax uses two of Racket’s interposition points: #%module-begin and #%plain-lambda. With these forms, language developers can simultaneously reuse Racket syntax and interpret it in a Video-appropriate manner.

Like #%app, #%module-begin is inserted at the start of every module and wraps the contents of that module. Hence, a re-implementation may easily implement context-sensitive transformations. In the case of Video, the re-implementation of #%module-begin lifts definitions to the beginning of the module and collects the remaining expressions into a single playlist.

Figure 43 shows the essence of Video’s #%module-begin syntax transformer. It is written in Racket’s syntax-parse language [Culpepper 2012], a vast improvement over the Scheme macro system [Bawden and Rees 1988; Clinger 1991; Dybvig et al. 1993; Kohlbecker et al. 1986; Kohlbecker and Wand 1987]. As before, the transformer is defined with
a different name, #%video-module-begin (line 6), and is renamed on export (line 3). The implementation of #%video-module-begin dispatches to video-begin (line 9), the workhorse of the module. This auxiliary syntax transformer consumes four pieces: an identifier (vid), a piece of code (export) formulated in terms of the identifier, a list of expressions (e ...), and the module’s body, which is represented as a sequence of expressions (body ...). In the case of #%video-module-begin, the four pieces are vid, (provide vid), (), and the module body.

The video-begin syntax transformer (lines 11–23) is responsible for lifting definitions to the beginning of the module body and accumulating expressions into a playlist. Its definition uses pattern matching again. Lines 13 and 17 specify the two pattern cases, one for when the module body is empty and another one that handles a non-empty sequence of body expressions:

- Once video-begin has traversed every piece of syntax (line 13), exprs contains all of the original module body’s expressions. The generated output (lines 14–16) defines the given vid to stand for the list of expressions bundled as a playlist.

- In the second case, the transformer expands the first term up to the point where it can decide whether it is a definition (line 18). Next, the transformer uses syntax-parse to check whether the elaborated code is a syntax list (lines 19 and 22) with a recognized identifier in the first position (line 21), in particular, define and provide.
– If so, the transformer lifts the first term out of `video-begin` and recursively calls itself without the newly lifted expression (line 21).

– Otherwise, it is an expression and gets collected into the `exprs` collection (line 23).

The astute reader may wonder about the generated `begin` blocks. As it turns out, Racket’s `#%module-begin` flattens `begin` sequences at the module top-level into a simple sequence of terms.

The syntax transformer for function bodies also uses `video-begin`. Instead of handing over `(provide vid)`, the call in the function transformer merely passes along `vid`, because functions return the produced `playlist`, they do not export it.

Figure 44 shows the syntax elaboration of a module using the Video specific `#%module-begin` form. The elaborated module describes the running conference talk example. Here, `#%module-begin^video` is Video’s variant, while `#%module-begin^base` is the one provided by racket.

### 22.3 VIDEO, BEHIND THE SCENES

Video relies on bindings to a C library to perform the rendering of video descriptions to files or streams. It exploits Racket’s FFI language for this purpose [Barzilay and Orlovsky 2004]. While Video uses the FFmpeg [Tomar 2006], any suitable multimedia library will do.

As it turns out, the Racket doctrine actually applies to the step of adding bindings too. The task of importing bindings manually is highly repetitive and developers frequently turn to other tools or libraries to construct the FFI bindings for them. Using a DSL to create the bindings has two advantages over using a library for a similar task. First, a DSL separates the task of constructing safe FFI calls with the task of connecting to a specific multimedia library. Using an FFI, developers can connect to a library safely by specifying the contracts. Second, a DSL allows developers to create new binding forms that are relevant to the multimedia library. It turns out that, once again, the effort of implementing this auxiliary DSL plus writing a single program in it is smaller than the effort of just writing down the FFI bindings directly. In other words, creating the DSL sufficiently reduces the overall effort so much that it offsets the startup cost, even though it is used only once.

The auxiliary DSL relies on two key forms. First form, `define-ffmpeg`, uses the Racket FFI to import bindings from FFmpeg. The second form, `define-constructor`, defines the core data types for Video and sets up a mapping to data that FFmpeg understands. While I chose to use FFmpeg to support Video, other rendering libraries can be used in its place. For example, early versions of Video used both GStreamer and MLT.
#lang video
(image "splash.png" (-elided-)
(conference-talk video (-elided-))
(define video (-elided-))

elaborates to
#lang video
(#%module-begin
(image "splash.png" (-elided-))
(conference-talk video (-elided-))
(define video (-elided-))

elaborates to
#lang racket/base
(#%video-module-begin
(image "splash.png" (-elided-))
(conference-talk video (-elided-))
(define video (-elided-))

elaborates to
#lang racket/base
(#%module-begin
(define video (-elided-))
(video-begin vid
(image "splash.png" (-elided-))
(conference-video video (-elided-)))

elaborates to
#lang racket/base
(#%module-begin
(define video (-elided-))
(provide vid)
(define vid
(playlist
(image "splash.png" (-elided-))
(conference-video video (-elided-))))

elaborates to

Figure 44: Compilation for a Video Module
The `define-ffmpeg` form is useful for hardening foreign functions. By using the `define-ffmpeg` form, programmers must only specify a contract [Findler and Felleisen 2002] that describes the valid inputs and outputs. Consider `ffmpeg_profile_init`, a function that initializes a C library. It takes a string and returns either a profile object or `NULL` if there is an error. Rather than having to manually check the input and output values, the FFI just states input and output types:

```
(define-ffmpeg ffmpeg-profile-init
    (_fun _string
        -> [v : _ffmpeg-profile-ptr]
        -> (null-error v)))
```

It errors if a bad input or output type passes through this interface.

The `define-constructor` form introduces both structures to represent video clips in Video and methods for converting these Video-level objects into structures that FFmpeg understands. For its first purpose, it generalizes Racket’s record-like `struct` definitions with an optional super-struct, additional fields, and their default values. For example, the following is the description of a Video-level producer:

```
(define-constructor producer service
    ([type #f] [source #f] [start -1] [end -1])
    (define producer* (ffmpeg-factory-producer (current-profile) type source))
    (ffmpeg-producer-set-in-and-out producer* start end)
    (register-ffmpeg-close ffmpeg-producer-close producer*))
```

This definition names the struct `producer`, specifies that it extends `service`, and adds four fields to those it inherits: `type`, `source`, `start`, and `end`.

For its second purpose, `define-constructor` introduces the body of a conversion function, which renderers know about. Here the function body consists of three lines. It has access to all of the structure’s fields, as well as the parent structure’s fields. The renderer calls this conversion code at the point when it needs to operate over FFmpeg objects rather than Video data structures.

### 22.4 The Preview Editor

Figure 45 shows the implementation for the preview extension discussed in the previous chapter. Rather than using a dedicated form for `define-react-component`, the renderer (line 2) directly lists a ClojureScript module and function that handles the previewing. The elaborator (lines 3–9), generates the resulting video code.

As is the standard in the bridge architecture, the renderer is defined in a ClojureScript function:
(define-interactive-syntax Video$
(render video.core Video$-render)
(elaborate stx
(syntax-parse stx
#:datum-literals (:timings :video)
[(_ (_ :timings [[start end] ...] :video video))
#'(playlist
  (playlist video #:start start
#:end end) ...)]))

Figure 45: Implementation for the Video Preview Extension

(defn Video$-render [this]
(let [timings (cursor this [:timings])
  video (cursor this [:video])
  [:> VideoEditor {:videoUrl (str server @video)
    :saveVideo ...elided...
    :timings (clj->js @timings)
    :updateState ...elided...}]}})

This render function hooks the state up to an existing VideoEditor library.
This library does not do any video rendering directly. Rather, it contacts a dedicated server running Video. That server evaluates the given expression and returns a preview video. This editor can then perform minor trim edits on the resulting preview.

The implementation for these interactive-syntax extensions is less than 100 lines of code, making use of several existing React libraries. For comparison, a pure Racket implementation of embedded NLVE widgets is roughly 800 lines of code, with 700 lines dedicated to just the graphics.
Part V

FINAL REMARKS
This dissertation draws on related work from several areas. This chapter covers these areas, and discusses the works from each. First, I discuss languages and environments that allow programmers to run custom programs as they edit code. Next, I discuss graphical and non-textual programming languages. After that, I touch on the concept of projectional and bidirectional editing. With these laid out, I move onto discussing Sandblocks, a system similar to the one presented here. Next, I move on to work related to the field of video editing. Finally, I discuss research related to human computer interaction and user evaluation.

### 23.1 Edit Time

Two rather distinct pieces of work combine edit-time computation with a form of programming. The first is due to Erdweg in the context of the Spoofax language workbench project and is truly about general-purpose programming languages. The second is Microsoft’s mixing of textual and graphical “programs” in the productivity suite.

Like Racket, Spoofax [Kats and Visser 2010] is a framework for developing programming languages. Erdweg et al. [2011] recognizes that, when developers grow programming languages, they would also like to grow the IDE support. For example, a new language feature may require a new static analysis or refactoring transformations, and these tools should cooperate with the language’s IDE. They therefore propose a framework for creating edit-time libraries. In essence, such libraries would connect the language implementation with the IDE and specifically the IDE tool suite. These libraries are IDE plugins and thus extra-linguistic.

Microsoft Office plugins, called VSTO Add-ins [Microsoft 2019], allow authors to create new types of documents and embed them into other documents. A developer might make a music type-setting editor, which another might use to put music notation into a PowerPoint presentation. Even though this tool set lives in the .NET framework, it is an extra-linguistic idea and does not allow developers to build programming abstractions.
Several programming systems have enabled a mixture of some graphical and textual programming for decades. The four most prominent examples are Boxer, Hypercard, Scratch, and Smalltalk.

Boxer [diSessa and Abelson 1986] allows developers to embed GUI elements within other GUI elements (“boxing”), to name such GUI elements, and to refer to these names in program code. That is, “programs” consist of graphical renderings of GUI objects and program text (inside the boxes). For example, a Boxer programmer could create a box that contains an image of a Tsuro tile, name it, and refer to this name in a unit test in a surrounding box. Boxer does not satisfy any of the other desiderata in chapter 4. In particular, it has poor support for creating new abstractions with regard to the GUI elements.

Scratch [Resnick et al. 2009], also an MIT product, is a fully graphical language system, with wide applications in education. In Scratch, users write their programs by snapping graphical blocks together. These blocks resemble puzzle pieces and snapping them together creates syntactically valid programs. Scratch offers limited, but growing [Harvey and Mönig 2010], capabilities for a programmer to make new block types. These created block types, however, are themselves created through text.

LabVIEW [Kodosky 2020] is another visual language, but targeted towards scientists and engineers. It has received wide adoption in its target communities. While possible to create robust products using labview, extending labview with new domains or types of visualizations is a non-trivial task.

Hypercard [Goodman 1988] gives users a graphical interface to make interactive documents. Authors have used hypercard to create everything from user interfaces to adventure games. While hypercard has been used in a wide variety of domains, it is not a general-purpose language.

Smalltalk [Bergel et al. 2013; Goldberg and Robson 1983; Ingalls et al. 2008; Klokmose et al. 2015; Rädle et al. 2017] supports direct manipulation of GUI objects, often called live programming. Rather than separating code from objects, Smalltalk programs exist in a shared environment, the Morphic [Maloney et al. 2001] user interface. Programmers can visualize GUI objects, inspect and modify their code component, and re-connect them to the program. No conventional Smalltalk systems truly accommodate general-purpose graphical-oriented programming as a primary mode, however, see section 23.4.

GRAIL [Ellis et al. 1969a,b] is possibly one of the oldest examples of graphical syntax. It allows users to create and program with graphical flow diagrams. Despite the apparent limitations of this domain, GRAIL was powerful enough to be implemented using itself.
23.3 BIDIRECTIONAL AND PROJECTIONAL EDITING

Bidirectional editors attempt to present two editable views for a program that developers can manipulate in lockstep. One example, Sketch-n-Sketch [Chugh et al. 2016; Hempel et al. 2018], allows programmers to create SVG-like pictures both programmatically with text and by directly manipulating the picture. Another example is Dreamweaver [Adobe 2019], which allows authors to create web pages directly and drop down to HTML when needed. Changes made in one view propagate back to the other, keeping them in sync. I conjecture that an interactive-syntax mechanism like ours could be used to implement such a bidirectional editing system. Likewise, a bidirectional editing capability would improve the process of creating interactive-syntax extensions.

Wizards and code completion tools, such as Graphite [Omar et al. 2012], preform this task in one direction. A small graphical UI can generate textual code for a programmer. However, once finished, the programmer cannot return to the graphical UI from text.

Projectional editing aims to give programmers the ability to edit programs visually. Indeed, in this world, there are no programs per se, only graphically presented abstract syntax trees (AST), which a developer can edit and manipulate. The system can then render the ASTs as conventional program text. The most well-known system is MPS [Pech et al. 2013; Voelter and Lisson 2014]. It has been used to create large non-textual programming systems [Voelter et al. 2012]. Unlike interactive-syntax extensions, projectional editors must be modified in their host editors and always demand separated edit-time and run-time modules. Such a separation means all editors must be attached to a program project, they cannot be constructed locally within a file. It therefore is rather difficult to abstract over them.

Barista [Ko and Myers 2006] is a framework that lets programmers mix textual and visual programs. The graphical extensions, however, are tied to the Barista framework, rather than the programs themselves. Like MPS, Barista saves the ASTs for a program, rather than the raw text.

The Larch Environment [French et al. 2014] also provides a hybrid visual-textual programming interface. Programs written in this

---

1 Intentional Software [Simonyi et al. 2006] has similar goals, but there is almost no concrete information in the literature about this project.
environment, however, do not contain a plain text representation. As a result, programmers cannot edit programs made in the Larch Environment in any other editor.

The Hazel project and Livelits [Omar et al. 2019] are also closely related to interactive-syntax extensions. Like editors, the Livelits proposal aims to let programmers embed graphical syntax into their code. In contrast to interactive-syntax extensions, which use phases to support editor instantiation and manipulation, the proposed Livelits will employ typed-hole editing. Finally, while the Livelits proposal is just a two-page blueprint, I conjecture that these constructs will not be deployed in the same range of linguistic contexts as interactive-syntax extensions (see chapter 5).

Eisenberg and Kiczales [2007] introduced an Eclipse plugin that brought graphical elements to Java. Like interactive-syntax, these graphical elements have a plain text representation, stored as Java annotations. This does mean that programmers can write code with this plugin, and view it in any plain-text editor. The main difference between this plugin and interactive-syntax extensions is the way these new types of extensions are created. This plugin requires extensions to satisfy a Java interface, with purely textual implementations. As a result, programmers cannot create meta-editors with this plugin.

23.4 Sandblocks

In February 2020, a research group (consisting of Leon Bein, Tom Braun, Björn Daase, Elina Emsbach, Leon Matthes, Maximilian Stiede, Marcel Taeumei, Toni Mattis, Stefan Ramson, Patrick Rein, Robert Hirschfeld, and Jens Mönig) at the Hasso-Plattner-Institut für Digitales Engineering at Universität Potsdam published a book-sized technical report [Bein et al. 2020] on the most closely related piece of research: the Sandblocks system.²

The Sandblocks system adds a form of visual interactive syntax to the Squeak implementation of the Smalltalk programming language. At first glance, the visual syntax extension look similar to the ones presented here. Its visual extensions can show up in the middle of ordinary program text and program text may show up inside of visual extensions. Developers can interact with these visual extensions and thus modify their appearance and state. Appearances are deceiving, however, and the differences are revealing.

Motivation The authors of Sandblocks share our motivation for extending Squeak with interactive visual syntax. Text is an easy-to-use medium for creating many—if not most—software, but a large number of thoughts expressed as linear text are much more easily comprehended via visual renderings. Since this insight immediately points

² The technical report is entirely in German. Since I do not read German, this section is based on my advisor’s notes from studying the Sandblocks book.
experts to the area of visual programming language, the *Sandblocks* book contains an extensive literature analysis. The analysis thoroughly explains the history and state of visual programming, points out its usefulness in certain situations, and focuses on its limitations.

These limitations are particularly visible in the area of computer-science education, where visual languages have made large inroads over the past two decades. While learners of all ages take to the visual programming idiom, they all eventually experience its “low ceiling” effect. That is, writing small, appealing graphical programs is easy; producing software systems or acquiring software development habits seems impossible. In particular, learners do not seem to be able to transfer the acquired skills to software development contexts and, as a result, scaling the programs to anything resembling a software system does not work.

The *Sandblocks* authors finally point to research that attempts to move programmers from visual programming to textual programming. Such “heterogeneous” languages [Erwig and Meyer 1995] do combine text and visual syntax but not in the manner of this dissertation or *Sandblocks*. Roughly speaking, these systems help programmers translate visual syntax into textual syntax, replacing ordinary linguistic statements (assignments, conditionals, loops, function calls) and expressions for visual renderings. This form of research seems to have ended in the early 2000s.

**Design** The design of *Sandblocks* aims at a form of interactive syntax that (1) is programmer-definable and (2) can replace any form of textual syntax.

Furthermore, (3) the designers insist that visual syntax may not interfere with any elements of the Squeak programmer’s tool chain. This tool chain includes the IDE in general. More specifically, the design chapter mentions code completion; copy and paste; search and replace; and several other such IDE functions. Furthermore, tools in the operating system must continue to work effectively: grep and its brethren, moving and copying; etc. The final element is code-repository and version-control system. Given developer’s heavy dependence on these systems, the designers declare it imperative that interactive visual syntax not interfere with those.

**Implementation** The implementation of *Sandblocks* falls short of the design with respect to goals (2) and (3). In the *Sandblocks* implementation, interactive visual syntax can replace only statements, including classes and nests of classes.

Squeaks’ Morph system is the key to implementing interactive visual syntax elements. Each element inherits its GUI capabilities from the Morph framework and implements two interfaces: one for interacting with developers at edit time and one for realizing functionality at run time:
• the Squeak IDE calls the element’s rendering and event-handling methods. These methods have access to the current edit-time state of relevant IDE elements. It is thus possible to, say, pop up a color-picker that records the desired choice of color in the program element and vice versa.

• the Squeak run-time system calls an execution method. It informs the method of the current execution state and allows the method to mutate the state as needed. As a result, the interactive syntax elements are unaware of their lexical context and implement a form of (Lisp-style) dynamic scope instead.

Finally, each visual syntax element implements a rendering method that serializes its current edit state into textual form. The IDE uses this serialization for all of its functions. The Sandblocks system can thus achieve all design goals with respect to the developer’s tool chain.

Illustrative Examples The Sandblocks book comes with a few small examples that illustrate the power of interactive visual syntax. The small examples include such simple functionality as the already mentioned color-picker. Two chapters presents large examples: a simple system for implementing the observer pattern via a UML-style notation and a visual toolbox for expressing graph-based (aka state-machine) flow of control (similar to the one presented in part III). In general, the examples replace existing (control) constructs but do not show how visual syntax can clarify domain-specific concepts.

Comparison While the Sandblocks design is a thorough document, it lacks a semantic perspective. Even an untyped programming language such as Smalltalk comes with a basic static semantics. In particular, Smalltalk programs and developers benefit from lexical scope for program identifiers. Unfortunately, the design description of “replace any form of textual syntax anywhere” comes without consideration for this critical aspect of the Smalltalk language. As a result, the implementation seriously compromises the underlying language, and by implication the developer’s toolchain.

Unlike the systems presented here, visual elements in the Sandblocks implementation cannot replace field definitions, methods, expressions, patterns, templates, and other syntactic forms. It is probably due to this limitation that the examples built in Sandblocks are pedestrian.

In summary, the Sandblocks design is ambitious and shares the motivation presented here. How seriously its one significant flaw interferes with a developer’s work remains to be seen. The implementation under-approximates the design in many regards. As a result, the evaluation comes up short. Nevertheless, the authors of Sandblocks and I aim to construct the same textual-visual language and are to be commended for their success.
People use so-called non-linear video editors (NLVEs) to compose video clips [Dancyger 2010]. In the context of film production, non-linear means non-destructive, that is, the source videos do not degrade in quality due to editing. A NLVE is a graphical tool with a time line of tracks. Each track describes a composition of video clips, audio clips, and effects playing in sequence. The NLVE renders these tracks by playing them all simultaneously, placing one track on top of the other. Obviously a screen displaying the result can play only a single track for video; by convention this is the last or top track. Video editors use effects to composite tracks. That is, effects splice two or more tracks together so that they appear on the screen at the same time. Technically, the renderer uses these effects to combine tracks as they play and either output the result to a file or play it on a screen.

Over time, professionals have developed tools and design patterns to reduce the amount of repetitive manual labor in video editing. They frequently develop so-called macros—a scripted sequence of user interface elements—in languages such as AppleScript [Cook 2007]. Some professional tools, such as Adobe Premiere [Jago and Adobe Creative Team 2017], even include an API to create script-style plug-ins directly. Extending tools in this fashion has limits. These kinds of macros are extremely brittle and frequently break, even within a single application, because these macro languages essentially specify dialog box clicks without understanding the underlying tools.

Using a tool’s official plug-in interface produces reasonably robust scripts but yields plug-ins that are tightly coupled with its tool. They can be used only when the entire toolchain is present. Blender [Roosendaal and Hess 2007], for example, is only scriptable with a Blender-specific Python interpreter that runs when Blender is launched.

Alternative approaches use general-purpose multimedia frameworks such as GStreamer [Taymans et al. 2013] or the MLT Framework. These frameworks are APIs for C-like languages that provide data types for building and rendering videos. These frameworks are primarily used in two situations. First, they are the back-ends to NLVEs. For example, MLT is the backend for both Shotcut and Kdenlive. Second, professionals use these frameworks to batch-process videos, particularly when interactive development is not desired.

The appeal of these frameworks comes from their ability to create abstractions, such as functions, to handle otherwise repetitive tasks.

---

3 Digital editors achieve this result by operating on references to videos, rather than operating on the videos themselves.
4 Audio tracks can actually be played simultaneously.
5 https://mltframework.org/
6 https://shotcutapp.com/
7 https://kdenlive.org/
Using these frameworks quickly becomes cumbersome, however, when there is a need to combine interactive and programmatic work flows, as is the case for the creation of conference recordings. Thus, studios tend to stick with NLVEs and use these frameworks only sparingly.

Professionals also use domain-specific languages for video editing. These DSLs primarily fall into two categories: XML-based DSLs and scripting-based DSLs. XML DSLs such as MLT XML and the now-deprecated SMIL [Bulterman et al. 2008] offer declarative languages for processing videos. These languages generally do not have functions or any other type of abstraction, however, and thus professionals tend not to deal with these XML languages directly when video editing. Rather, NLVEs use these languages as a file format to save video projects.

Scripting-based DSLs such as AVISynth\footnote{https://avisynth.nl} are declarative and support functions and other abstractions, but have their own limitations. They typically support only the simplest of tasks such as playing videos in sequence with transitions and minor visual effects. They also tend to lack any formal grammars and use a small script for a parser. Finally, they lack many code reuse features. AVISynth, for example, allows programmers to create simple functions but comes without any control flow constructs such as conditional branching.
FUTURE WORK

There is still more work to be done in the space of visual and interactive syntax. At a surface level, the current core.visr prototype can benefit from more engineering to polish the user experience. Going deeper, however, shows that interactive syntax itself still has many unanswered research questions. This chapter lays out the future research questions that this dissertation raises.

24.1 EDIT-TIME CODE AND PHASING

This dissertation introduces an ad-hoc definition of edit time. That is, code that runs as the programmer edits a program. While this lax definition suffices for the research presented in this dissertation, it limits the range of possibilities that arise from a precise definition.

One possible next questions is: do edit-time phases compose in the same way that compile-time phases do [Flatt 2002]? In languages with meta-programming systems, such as Racket, users write code that runs at compile time, and code that runs at that compile time’s compile time, and so on. The obvious question, should and does edit-time code stack the same way? Also, how does edit-time code intermix with compile-time code in a stacked fashion? Does it make sense to have edit-compile-time code, and if so, how does it differ from compile-edit-time code?

In addition to the stacking of edit-time phases, are there other phase types? Some simple examples are, test-time phases, documentation-time phases, or even debugger-time phases. Individual domains might even have their own notion of phases.

From a theoretical perspective, phases might be similar to free groups [Stillwell 1951]. Using this model, each phase would have an anti-phase, and each module would be required with respect to an absolute phase-path. A phase and its anti-phase would cancel each other if they were immediately next to each other in the path. For example, compile-anticompile would cancel, while compile-edit-anticompile would not. Initial sketches show that this is a logically consistent system, but is currently unclear if this system is worth the added programming complexity.

24.2 USABILITY, USEFULNESS, AND PRACTICALITY

Another area of concern involves the usefulness and usability criteria of interactive-syntax extensions. The study conducted for this dis-
sertation indicates only that interactive-syntax extensions are usable. But, the small population size raises the question of what makes an interactive-syntax extension mechanism truly useful and usable.

One way to assess the value of visual-interactive syntax is to conduct a long-term investigation. Given that my goal is to raise the quality of messaging from one developer to another over time, the primary question is whether and how visual syntax improves a developer’s comprehension of old program (snippets) in a large code base. To this end, visual interactive syntax must become easy to create, easy to use, and ideally usable in a wide variety of IDEs.

To validate this last point, I imagine that research into this question would start with a corpus analysis. This analysis would first determine where and how interactive syntax libraries exist, similar to studies analyzing existing programming languages’ package systems [Morandat et al. 2012]. Second, it would have to evaluate how many developers use these interactive-syntax extensions [Berger et al. 2019].

Once the results of such a quantitative analysis are available, researchers could then set up an observation of developers who modify pieces of software that contain visual syntax. The goal of the observation would be to check how quickly developers understand the code to the point where they can modify or add code. For example, when following best practices, a developer fixing a bug would have to write failing unit tests. In a context with visual unit tests, such as the ones for the running Tsuro example, I would expect the developer to create a new unit test in a short amount of time—and that would supply additional evidence in support of visual syntax. In the ideal case, researchers would also identify pieces of code of similar complexity in the same code base that do not come with visual syntax and observe how quickly a developer could articulate a unit test in such a context. My conjecture is that the time spent in the latter case is vastly longer than in the former. If research can validate this conjecture, it would simultaneously verify the usefulness and usability of my ideas.

A secondary goal of a longitudinal study is to understand best practices for creating new types of interactive-syntax extensions and for deploying existing visual-syntax libraries. The two questions to address are (1) when developers should take the time to develop a new extension and (2) when developers should decide to look for a suitable library and switch from text-only to heterogeneous coding. My anticipated answers are that areas of coding that need geometrical (trees) and topological (networking) thinking, call for libraries of easily composable pieces; and every developer who draws a diagram on paper or an ASCII diagram in a file should consider using visual syntax. Scholars would translate validated answers to these questions into instructional material for students at universities and working developers in companies.
The prototypes and user studies in this dissertation validate that it is indeed possible to construct a usable and useful mechanism for programmatically adding visual and interactive, domain-specific syntax to an existing textual programming language. The first part shows the usefulness of interactive-syntax extensions. The second part verified that they can also be usable. Next, the third part extends the usability to languages that could benefit from interactive-syntax extensions, but cannot support them natively. And finally, the last part provides an extended case study of interactive-syntax in practice.

Visual and Interactive Syntax, as presented in this dissertation, builds off of the work of many attempts at allowing programmers to write code in non-textual mechanisms. Interactive-syntax extensions are the first approach that allow programmers to:

- mix text and code;
- keep a reasonable coding experience in any text editor or IDE; and
- extend the available set of interactive visual features.

Additionally, my approach is a blueprint for equipping existing programming languages with an extension mechanism. In short, it offers a truly linguistic solution.

Finally, I promised myself I would put something silly somewhere in my dissertation, so enjoy this picture of a cake using interactive syntax:


Hubert Comon, Max Dauchet, Remi Gilleron, Florent Jacquemard, Denis Lugiez, Christof Löding, Sophie Tison, and Marc Tommasi. Tree Automata Techniques and Applications. 2007.


139
Mary Shelley. Frankenstein; or, The Modern Prometheus. Lackington, Hughes, Harding, Mavor & Jones, 1818.


