Making Hybrid Languages: A Recipe

LEIF ANDERSEN, University of Massachusetts Boston, USA
CAMERON MOY, Northeastern University, USA
STEPHEN CHANG, University of Massachusetts Boston, USA
MATTHIAS FELLEISEN, Northeastern University, USA

The dominant programming languages support only linear text to express ideas. Visual languages offer graphical representations for entire programs, when viewed with special tools. Hybrid languages, with support from existing tools, allow developers to express their ideas with a mix of textual and graphical syntax tailored to an application domain. This mix puts both kinds of syntax on equal footing and, importantly, the enriched language does not disrupt a programmer's typical workflow. This paper presents a recipe for equipping existing textual programming languages as well as accompanying IDEs with a mechanism for creating and using graphical interactive syntax. It also presents the first hybrid language and IDE created using the recipe.

1 MIXING TEXT WITH VISUAL AND INTERACTIVE SYNTAX

Programmers use programming languages to communicate their thoughts, both to computers and to other programmers. Linear text suffices for this purpose most of the time, but some thoughts are inherently geometric and better expressed visually.
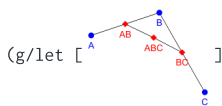
Recognizing this problem, researchers have devised many solutions ranging from purely visual languages [Resnick et al. 2009], to special-purpose IDEs [Perez and Granger 2007], and various other strategies [Bein et al. 2020; Omar et al. 2021]. In particular, Andersen et al. [2020] proposed the idea of a *hybrid language*, which combines textual code with miniature graphical user interfaces (GUIs), dubbed visual and interactive syntax. Using a hybrid language, programmers can communicate their thoughts with text most of the time and weave in visual interactive constructs when it is appropriate for the problem domain.

For example, consider the calculation of quadratic Bézier curves [Farin 2014]. The standard algorithm combines two tasks: finding midpoints and recursion. While the second is easily expressed via text, the first is a geometric idea that deserves a pictorial representation. The build-bez function in figure 1 illustrates how a programmer might use a hybrid variant of Clojure to convey these two ideas. It computes a set of Bézier points from three input points that form a triangle. While the embedded visual syntax depicts the midpoint (of midpoints) calculation, recursively computing the rest of the points remains textual.

```
;; A Point is: {:x Real :y Real}

;; Point Point Point -> #{Point ...}
;; Compute the set of Bezier points
(defn build-bez [A B C]

  (g/let [                    ]

  (if (close-enough? A B C)
    #{A C ABC}
    (union #{ABC}
           (build-bez A AB ABC)
           (build-bez C BC ABC)))))
```

Fig. 1. Computing Bézier points

Unfortunately, existing attempts at visual syntax have fundamental flaws that either unreasonably disrupt a programmer's typical workflow, impose an undue burden on a language implementation, or both. For example, such solutions almost always force programmers to use one particular (new) IDE for their work, a non-starter for most. Andersen et al. [2020]'s solution forces a language

implementation to maintain *two* different GUI libraries and keep them synchronized—when even one such library already imposes a serious amount of work for most languages. This duplication makes the implementation costly to unmaintain. It also means programmers will often have to implement the same GUI twice: once for the actual user interface—using the original GUI library—and a second time for the visual syntax renderer—using the special-purpose library.

This paper's main contribution is the development of a general recipe for creating *maintainable* and *usable* hybrid languages. All prior work lacks at least one of these attributes. A key idea is that hybrid languages should be created by adapting existing languages and IDEs instead of creating new ones. Doing so improves usability because it allows programmers to keep using a familiar language, and a popular IDE, when programming with the hybrid language. It also helps maintainability because it reuses a language's existing infrastructure and libraries. To show this concretely, a second contribution of this paper is to apply the recipe to create Hybrid ClojureScript and a compatible CodeMirror-based hybrid IDE.

More specifically, section 2 describes the design goals of the recipe, which come from studying existing solutions. Next, section 3 presents the ingredients needed for the recipe: an existing programming language, a general-purpose IDE, and a GUI library. Section 4 provides a quick glimpse at the result of applying the recipe to ClojureScript and other chosen ingredients. The subsequent two sections explain the recipe in detail. Specifically, section 5 explains how to adapt a language to support hybrid syntax, and section 6 explains how to adapt an existing IDE so that it may visualize hybrid syntax and allow programmers to interact with it. Next, section 7 compares various hybrid languages and IDEs along several dimensions, and section 8 describes several case studies. Finally, the last two sections explain related work and conclude.

2   REQUIREMENTS FOR CREATING A HYBRID LANGUAGE

An examination of existing work suggests the following guidelines for a hybrid-language recipe:

(1) The goal must be to adapt an existing language. Doing so immediately makes the resulting language usable because programmers can continue working in a familiar context. The result is maintainable because implementers do not need to work on new system.

(2) Likewise, hybrid IDEs should be adapted from existing ones. This comes with the same advantages for IDEs as for the languages.

(3) The hybrid language must be backwards compatible, meaning that it can run existing non-hybrid code, and that hybrid programs can run on non-hybrid implementations.

(4) All existing IDEs and text editors must work with hybrid code. Since programmers have strong IDE preferences, forcing a specific IDE choice imposes a serious usability burden on programmers. Hybrid code should also remain compatible with other existing tools.

(5) Implementations of visual syntax must be able to re-use existing GUIs and GUI libraries. This relieves language implementers from having to maintain two different GUI libraries. It also helps programmers avoid duplicate effort, because they will often create interactive-syntax extensions that are identical to the GUI found in the application's run-time code.

(6) Finally, visual and interactive syntax should be linguistic, meaning it must smoothly integrate with textual syntax and all the language's abstraction mechanisms. Similarly, visual syntax should not just be a new category of syntax, but it should ideally be possible to turn all existing syntactic categories of the chosen language into visual syntax constructs.

Section 9 presents a more detailed analysis of the most closely related pieces of research and how each of them lives up to the above guidelines.

3 THE INGREDIENTS

Every recipe starts with a list of ingredients. As described in the last section, the basic ingredients for making a hybrid language are (1) an existing programming language; (2) an IDE, and (3) a GUI library. Since different quality ingredients can affect the outcome of a recipe, this section describes some additional attributes that facilitates the construction of hybrid languages and IDEs, and also makes the results truly usable.

3.1 Selecting High-Quality Ingredients

Ideally, the chosen language comes with a syntax-extension mechanism. Since the goal is to add interactive syntax for any problem domain, a programmer needs a mechanism for interpreting new syntactic features in the language. While it is possible to create shallow embeddings of domain-specific notations in any language, a syntactically extensible language greatly facilitates this step.

Similarly, the chosen IDE should (1) provide an extension interface for plug-ins and (2) support the execution of code at edit time. By using a plug-in tool, it becomes possible to interpret visual syntax extensions as mini-GUIs in the IDE's editor. Since this interpretation must run while the programmer edits code, an IDE that can run code at edit time in an isolated fashion is the best match. After all, programmer-created code may accidentally interfere with a logical invariant of the IDE's implementation if it is run without protection. Of course, this edit-time code must simultaneously be written in the chosen language and must cooperate with the IDE's editor—which suggests additional constraints on the chosen GUI library.

Besides being suitable for building application-level graphical interfaces, the GUI library must come with a text editor that is the same as the chosen IDE's editor. Furthermore, the editor must allow the insertion of GUI widgets (canvases, buttons, menus). By meeting these two criteria, programmers should easily be able to share run-time GUI code with edit-time GUI syntax extensions.

3.2 Example Ingredients

Given these attributes, ClojureScript, CodeMirror, and the DOM (Document Object Model) are reasonable choices. ClojureScript supplies a Lisp-style macro system that makes it easy to create a construct for defining visual and interactive syntax. The CodeMirror IDE comes with a rich plugin API, though it does not inherently isolate code that runs at edit time.

As for the GUI attribute, ClojureScript is a scripting language for the DOM, and CodeMirror's editor is based on the DOM as well. These ingredients can smoothly collaborate at edit time, as long as the IDE can be protected from problems in the programmer-supplied code for interactive syntax extensions.

Of the three ingredients, the DOM is particularly beneficial. It is standardized and can be found at the heart of web browsers, modern IDEs such as Visual Studio Code, and even some native operating systems like Android. Three decades of development work have turned it into a performant and expressive technology. At the same time, programmers have contributed a large collection of widely available GUI libraries for the DOM such as React, Angular, and Vue.js. All of these are familiar to many programmers and can thus be used to quickly build both application-level GUIs and re-used for the creation of interactive syntax extensions.

4 A FIRST TASTE

Tasting a dish is often incentive to ask for the recipe. In this spirit, this section provides a brief overview of HYBRID CLOJURESCRIPT and its use in elIDE, the hybrid CodeMirror-based IDE.

Figure 2 displays what a programmer sees when programming in HYBRID CLOJURESCRIPT using elIDE. Concretely, the screenshot shows the code of the Bézier curve function from figure 1 in
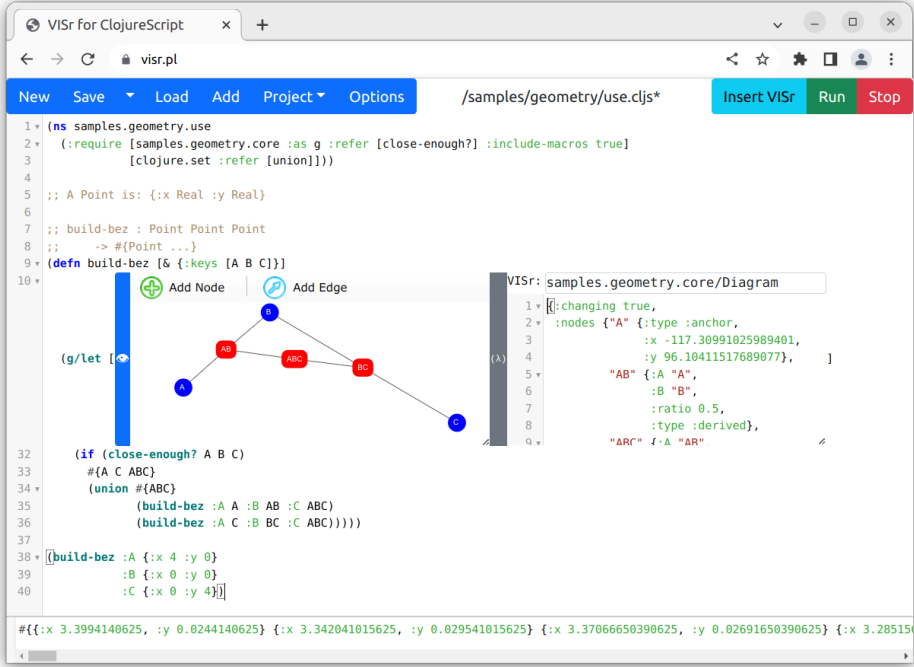
Fig. 2. In-IDE view of the Bézier function

elIDE. The most interesting part of the screenshot is the use of the interactive-syntax extension described in the introduction.

As the screenshot shows, the implementation of visual syntax for the Bézier function renders the code in two ways: a visual view on the left and a plain textual one on the right. The visual view generalizes the standard diagrams for midpoint calculations that students might see in a geometry class. The diagram is to be understood abstractly, meaning it computes midpoints based on the run-time position of the given nodes (A, B, and C) and their relative position to each other.

Equally important is the purely textual representation of the code seen right next to the visual view. This text is what the IDE puts into a file when the programmer saves the code. Hence any ClojureScript implementation can run this saved version of hybrid code. Better still, text is what IDE tools or command-line tools process, which implies that programmers continue to benefit from all these tools as they develop in the hybrid language. Finally, the text is also what programmers see when they open the code in unadapted IDEs or plain-text editors.

Another important aspect of interactive syntax is that extensions implement a model-view-control pattern. That is, a change to either view is immediately reflected in the other one. A model—dubbed the state—reconciles the two views with each other. When a programmer uses gestures to manipulate the mini-GUI, the implementation of interactive syntax changes the state; the IDE notices the change and updates both views.

Figure 3 provides a close look at the plain-text view of an interactive-syntax construct, which is just a function application. Specifically, it applies a function from the definition of the interactive-syntax extension to a textual version of the current state. Here the reference is to the Diagram extension, found in the geometry.core module.

```
^:visr
(geometry.core/Diagram
  {:changing false,
   :nodes
     {"A" {:type :anchor, :x 0, :y 20},
      "AB" {:A "A",
            :B "B",
            :ratio 0.5,
            :type :derived},
      "ABC" {:A "AB",
             :B "BC",
             :ratio 0.5,
             :type :derived},
      "B" {:type :anchor, :x 40, :y 0},
      "BC" {:A "B",
            :B "C",
            :ratio 0.5,
            :type :derived},
      "C"
        {:type :anchor, :x 50, :y 70}}}})
```
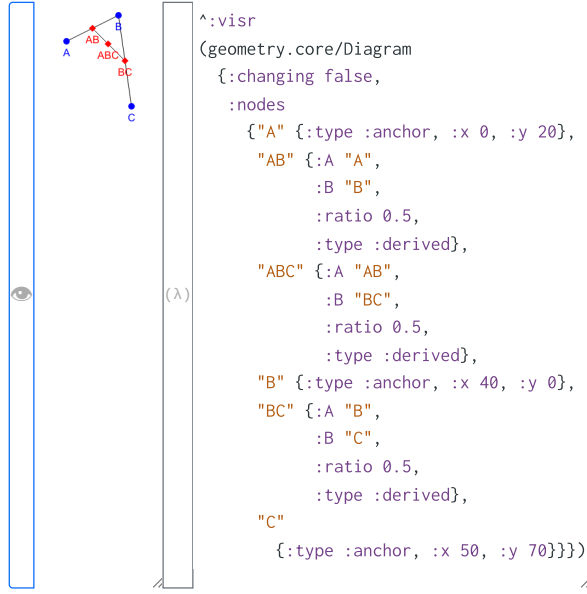
Fig. 3. A close look at the Bézier-specific syntax extension

This examination of the textual view demonstrates two points. First, a programmer can create an instance of an interactive-syntax extension in a plain text editor. No special IDE is needed. Second, a programmer can change the reference pointer to the interactive-syntax definition (Diagram), and the visualization on the left would change immediately. (If elIDE cannot find the implementation, it falls back on a default view.)

Even this brief tour validates how HYBRID CLOJURESCRIPT running in elIDE satisfies all desiderata of section 2. Table 4 contrasts the system with the work of Andersen et al. [2020], the closest competitor. It clarifies how the system presented here cooperates with IDE tools properly and preserves the existing workflow. Further, the edit-time GUIs use the same library as run-time GUIs and the IDE itself. Besides making the language and IDE easier to use, it also makes HYBRID CLOJURESCRIPT running in elIDE have far better performance characteristics than Andersen et al.'s adaptation of Racket and DrRacket.

5 A RECIPE FOR ADAPTING A LANGUAGE

If an existing programming language comes with bindings for an appropriate GUI library, then turning it into a hybrid one can happen in a step-by-step fashion. This section explains those steps, illustrates them with ClojureScript, and presents a complete example in the hybrid variant.

5.1 The Recipe

A hybrid language allows programmers to add new, problem-specific syntactic constructs to the already-available vocabulary. Programmers can then use these constructs to build libraries or full programs with interactive and visual syntax.

Given this context, the first step of the recipe requires creating syntax for defining new kinds of interactive syntax. More specifically, this new definition form specifies how interactive-syntax extensions keep track of their *state* (of the model), i.e., those values that must persist; how they *render*

Property	Related Research Language GUI library	Andersen et al. [2020] Racket bespoke GUI	this paper ClojureScript DOM
Add Interactive Syntax to Existing, Textual PL	✓		✓
Adapt Popular, General-Purpose IDE		✗*	✓
Hybrid PL is backwards compatible		✓	✓
Hybrid IDE is backwards compatible		✗†	✓
Standard GUI library, GUI Component Reuse		✗‡	✓
Linguistic visual and interactive syntax		✓	✓

*   Limited to its hybrid capabilities when used in an IDE.
†   Standard IDEs are forwards compatible, only hybrid IDEs break compatibility.
‡   Some reuse is possible by using a shim to generate GUIs from a common source.

Fig. 4. Desiderata comparison for interactive syntax designs

this state as a mini-GUI, or *serialize* it as plain text; and how these mini-GUIs *react* to programmer gestures that, in turn, manipulate the state. This setup closely follows the MVC architecture. Finally, when the programmer wishes to run programs constructed with interactive visual syntax, the interactive-syntax extension must know how to *elaborate* the textual view to a run-time semantics; this may happen via a "compilation" or an "interpretation".

The visible novelty here is that the state of interactive-syntax extensions can be rendered as either a mini-GUI for use in adapted IDEs or plain program text. The latter is used in both adapted and non-adapted IDEs, as well as when the entire program is run. Additionally, when the programmer interacts with the code in either modality, the interactive-syntax state must be updated so that both views show the up-to-date rendering as needed.
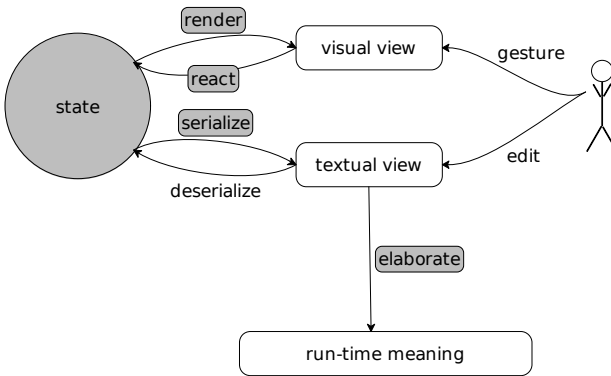

Fig. 5. Interactive-syntax extensions at work

Figure 5 sketches how an interactive syntax instance responds to programmer stimuli. It highlights (in gray) the five elements that the creator of an interactive-syntax extension must specify with the definition form.

The challenge for hybrid-language implementers concerns program phases. While programming always involves three phases—*edit time*, when the programmer edits the code; *compile time*, when the code is compiled; and *run time*, when the resulting target code runs—interactive-syntax extensions demand that *programmer-defined* code can run at edit time and compile time.

For a language implementer to follow this recipe means picking a representation for instances of an interactive-syntax extension—ranging from strings (bad) to algebraic data types (acceptable) to S-expressions (fantastic, due to its synergy with the multi-phase reflective nature of interactive-syntax extensions)—and to supply an interpreter or a compiler for this notation. In the context of JavaScript, for example, a "cook" could use a transpiler framework to assign semantics to new syntactic elements. In the context of a macro-extensible language, however, the work is even simpler; it suffices to implement a single (but non-trivial) macro definition. The next subsection illustrates this particular technique by applying the recipe to ClojureScript.

5.2 Applying the Recipe to ClojureScript

In order to turn ClojureScript into a hybrid language, it suffices to define a single macro, named defvisr, whose purpose is to define new interactive syntax extensions. To use defvisr, a programmer must specify three elements:

(1) a state element, which is an association of field names with initial values;
(2) render, which equips the extension with edit-time semantics; and
(3) elaborate, which assigns run-time semantics to the current state.

Here is a template of the new definition form:

```
1  (defvisr Name
2    (:state field-name field-value ...)
3    (:render [this] ... rendering code ...)
4    (:elaborate [this] ... elaboration code ...)
```

A defvisr definition introduces a new interactive-syntax construct, which can be instantiated many times, via plain text code or via GUI gestures (in a hybrid IDE). The state component specifies the state part of the interactive-syntax extension, as indicated by the "state" box in figure 5.

The use of defvisr specifies three computations. First, render consumes one argument—named this by convention—which is the current state. It turns the state into a DOM element that is sent to the IDE, assuming it is suitably adapted. Following standard DOM-development practice, render collapses view and control. That is, it is simultaneously responsible for drawing the GUI and for handling user input that allows the direct manipulation of the state. For the second aspect, render may mutate the fields of the state. In ClojureScript terminology, the defvisr macro implementation supplies render with an "atom" containing the state. Thus, the render component implements the "render" and "react" boxes from figure 5.

Second, the extension provides serialization for states using JavaScript's serialization facilities, as required by the "serialize" box in figure 5. Observe, however, that a defvisr definition does not require specifying serialization explicitly. Instead, a defvisr instance implements this functionality implicitly for the programmer.

Third, like render, elaborate consumes the current state (as text) as its sole argument. Its task is to interpret the serialized state when the ClojureScript program runs, as indicated by the "elaborate" box in figure 5. The textual view expresses this idea with a call to the elaborator (which actually has the same name as the interactive-syntax construct itself, e.g., the Diagram defvisr defined below) wrapped around the serialized state. This expression may end up being a function application or, since this is ClojureScript, a macro. In the latter case, elaborate may generate

compile-time code, which can, for example, set up new variable bindings or statically check instances of the interactive-syntax extension.

As a convenience, defvisr exploits the state specification to simplify the syntax of the rendering and elaboration code. Specifically, it implicitly binds the names of the fields of the state in the scope of the two function bodies for reference. Mutation must use ClojureScript's atom functionality.

```
1  ;; A DIAGRAM is: {:nodes {<STRING NODE> ...} :changing BOOLEAN}
2
3  ;; A NODE is one of:
4  ;; - {:type :anchor :x NUM :y NUM}
5  ;; - {:type :derived :A STRING :B STRING :ratio NUM}
6
7  (defvisr Diagram
8    (:state nodes    {}
9           changing true)
10   (:render [this] (render-state-as-dom-element nodes changing)))
11   (:elaborate [this] (elaborate-diagram-to-syntax nodes))
12
13 (defmacro (g/let [diagram] & body)
14   `(clojure.core/let ~(macroexpand diagram) ~@body))
```

Fig. 6. The defvisr for the interactive-syntax extension in figure 1

5.3 Working with the Adapted ClojureScript

Figure 6 sketches a defvisr definition of the midpoint extension used in figure 1.

State. As the comments explain, the Diagram extension manages values of type DIAGRAM. That is, the state consists of some nodes and a boolean flag, called changing. Each NODE contains its type and its position; the nodes field manages the NODEs and their connections. The information in nodes is used for drawing the diagram at edit time as well as setting up variable bindings for the body of the plain-text g/let macro at compile time. The changing field of the state is set to true when the programmer is actively modifying the diagram; it is an edit-time only value.

The :type field indicates that there are two distinct classes of nodes. *Anchor* nodes are inputs to instances of the Diagram interactive-syntax extension; their positions become known at run time only. In the example from figures 2 and 3, A, B, and C are anchor nodes. *Derived* nodes are outputs of the midpoint calculation; their values are determined algebraically from anchor nodes and other derived nodes. In the previous example, AB, BC, and ABC are derived nodes. An interaction with the visual diagram could shift these derived nodes and assign weights other than 0.5, yielding a different kind of curve calculation.

The Renderer. Figure 7 sketches the renderer implementation for the Diagram interactive-syntax extension. Concretely, the code on the left side of the figure shows the function, Diagram-view, which renders a Diagram as a GUI view. It reuses functionality from a runtime GUI library and is thus straightforward for creators of the interactive syntax to write. More specifically, an external JavaScript library, visjs, handles the low-level drawing and event handling for the Diagram. The :> (line 6 on the left) is a special keyword that acts as a foreign function interface (FFI) to external JavaScript libraries.

The code on the right side is the functionality needed to use this library code for the actual Diagram renderer. The require specification imports the library, in particular, the Diagram-view function. As mentioned, the render-state-as-dom-element function is applied to an atom that

```
1  (ns geometry.library
2      (:require [visjs]))
3
4  ;; (Atom #{Node ...}) -> Dom-Element
5  (defn Diagram-view [nodes]
6    [:> visjs/Graph {:options ...elided...
7                     :events ...elided...
8                     :graph (build-diagram @nodes)}])
```

(a) `library.cljs`

```
1  (ns geometry.elaborator
2      (:require [geometry.library
3                 :refer [Diagram-view]]))
4
5  ;; (Atom #{Node ...}) (Atom Boolean) -> DOM-Element
6  (defn render-state-as-dom-element [nodes changing]
7    [:div {:style ... elided (uses changing) ...}
8     [Diagram-view nodes]])
```

(b) `renderer.cljs`

Fig. 7. Renderer for a geometry extension

contains the state. An atom in ClojureScript is essentially a mutable box. From this state, the renderer computes a data structure that encodes the user-facing DOM-element. Functions placed in the first position in a vector (e.g. `Diagram-view` on line 8) are treated as sub-components to be rendered. Likewise, keywords (e.g. `:div` on line 7) directly represent DOM tags.

This rendering code is also called in response to a programmer's interaction with the mini-GUI. It then reads and modifies the state through interactions with the state atom. Unboxing the atom, through the `@` operator (on the left), returns an immutable encoding of the state (line 8, left). When the state atom changes, a publish–subscribe style watcher (in the `visjs` library) notices and updates the two views.

```
(g/let [(Diagram {:nodes {... ...}
                  :changing false})]
  body ...)
```

$\Longrightarrow$
elaborates

```
(let [{:keys [AB BC ABC]}
      (compute-mid-points ...nodes...
                          {:A A :B B :C C})]
  body ...)
```

Fig. 8. Elaborator for a geometry extension

The Elaborator. Generally speaking, an interactive-syntax elaborator is a syntax-to-syntax function. Figure 8 shows an example of how `Diagram`'s elaborator (which is invoked by directly applying the `Diagram` name to a state representation) is used. As seen on the left-hand side, this particular `Diagram` interactive-syntax is designed to be used with a special `g/let` macro. Specifically, elaboration of a `Diagram` produces two parts: a sequence of identifiers (called keys in ClojureScript), and an expression.

The right side of the figure shows how these elaboration results are used in the run-time representation of the program: the three computed identifiers are used as binders in a plain `let`, and the expression computes the values for these binders. More specifically, the expression is an application of `compute-mid-points`—a run-time function—to symbolic names and the concrete anchor nodes; it computes the derived node positions based on the run-time position of the anchor nodes. All of this sets up the three variable bindings that are seen as the three red midpoint dots in the visual interactive syntax from figures 1 and 2.

6 ADAPTING AN IDE TO HYBRID CLOJURESCRIPT

A hybrid programming language must come with at least one IDE that can display interactive-syntax extensions visually and textually. This section describes elIDE, an adaptation of CodeMirror for writing programs in HYBRID CLOJURESCRIPT. CodeMirror is a DOM-based editor that serves as the foundation of a number of IDEs.

6.1 `elIDE` Needs a Hint

A closer look at figure 3 (on page 5), specifically the textual view on the right, reveals an explicit
`^:visr` Clojure metadata prefix on the first line. This tag tells `elIDE` that (1) this expression is an
instance of interactive syntax and (2) the expression contains a reference to the implementation
for its visualization. This use of metadata plays a key role in getting an adapted IDE to work with
hybrid syntax.

6.2 The Architecture of `elIDE`

Unlike an IDE for plain-text programming, an IDE for a hybrid language must run *user-defined code*
at edit time. Enabling an IDE to run user-defined code at edit time raises a number of concerns,
most importantly, that edit-time code may interfere with the integrity of the IDE. Also, any design
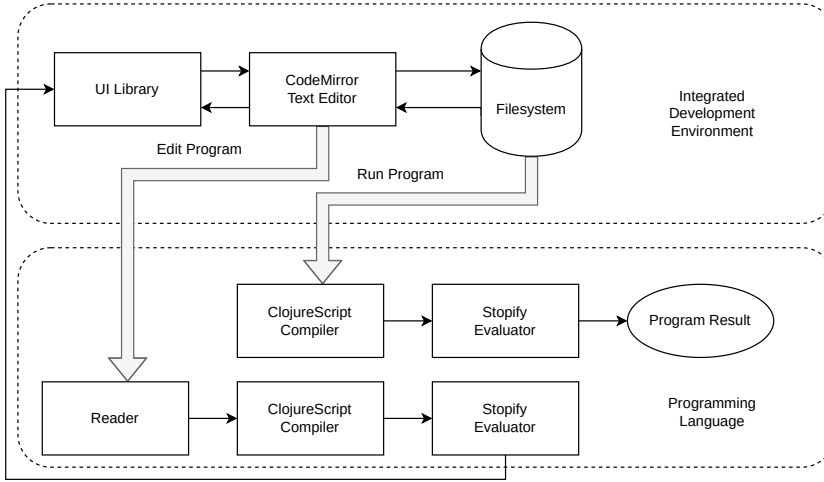must be modular so that the hybrid language and the hybrid IDE do not become tightly coupled.

Fig. 9. Architecture for `elIDE` (top) and Hybrid ClojureScript (bottom)

Figure 9 shows an architecture diagram of `elIDE` (top) and Hybrid ClojureScript (bottom). The
rest of this subsection first explains the standard IDE features and then those added for Hybrid
ClojureScript.

Standard IDE Facilities. The `elIDE` IDE supports the expected functionality: (1) editing code with
the CodeMirror text editor; (2) storing code in a file and loading it from there;[1] (3) running code;
and (4) background execution. All of these facilities are realized in the expected fashion.

Hybrid IDE Facilities. The key hybrid facility is to reflect changes to the state of an instance of
an interactive-syntax extension in the program's displayed source code. Whether these changes
happen via text editing or direct manipulation of the GUI does not matter. By allowing Hybrid
ClojureScript to communicate back to `elIDE`, this reflection is enabled in a natural manner.

Adding interactive syntax to a language means that a visual IDE must continuously run the
rendering code of instances of interactive syntax in the background. As a programmer edits inter-
active syntax, the IDE updates the state if needed and calls the appropriate functions to present
the visualization and the plain-text variants as needed.

---

[1]Because `elIDE` runs in a browser, it uses BrowserFS [Powers et al. 2017], a lightweight file system for browsers.

A HYBRID CLOJURESCRIPT specific reader (figure 9) is the entry point for this task. It scans the entire program, determines which portion of the code runs at edit time, and sends this portion of the code to the background evaluator.

In order to satisfy the edit-time evaluation requirement, ClojureScript is bootstrapped with an evaluator that composes the ClojureScript compiler and Stopify [Baxter et al. 2018]. The latter is a JavaScript transpiler and run-time environment whose purpose is to compile straight JavaScript into code that supports cooperative multitasking through continuation passing. In the context of HYBRID CLOJURESCRIPT, Stopify supplies two pieces of functionality. First, it allows the IDE to pause running programs, i.e., misbehaving interactive-syntax extensions do not lock up the IDE. Second, it provides a sandbox environment that separates edit-time code for interactive-syntax extensions from the IDE. It thus prevents the former from interfering with the IDE's internals.

Finally, the rendering code of interactive syntax is the only way for HYBRID CLOJURESCRIPT to send information back to the IDE. State managed in this code is translated back into the program as text. Specifically, the implementation turns the required changes into code snippets written in a standard JavaScript library for manipulating the DOM. (The right-hand side of figure 7 shows a concrete example of what these snippets look like and how they are computed.) When these pieces of code are sent to CodeMirror, they place DOM elements into the text editor at the proper places.

6.3 The General Idea

The lessons learned from building a hybrid IDE with CodeMirror generalize to a recipe. It starts from an IDE that uses the same GUI library, including an editor, as the hybrid language. To make the IDE hybrid, that IDE must be able to collaborate with hybrid languages so that it can display instances of interactive syntax as mini GUIs. This collaboration covers three aspects:

(1) the language implementation can tell the IDE which pieces of the code are interactive;
(2) the IDE can request that the language implementation evaluate the GUI code from identified instances of interactive syntax at edit time and complete programs at run time; and
(3) the language implementation may insert elements into the IDE's editor.

Without the last, the IDE cannot show programmers interactive syntax as GUIs instead of text.

Hence, adapting an existing IDE to a hybrid language is easily implementable if it comes with a plug-in API or a similar capability. This plug-in API must grant full access to the IDE's editor, and it must support callbacks that allow the IDE to invoke the language implementation on both pieces of functionality—at edit time—and complete programs—at run time. The first kind of callback relies on the above-mentioned full access capability so that it can insert the results of the evaluation at the appropriate places. Finally, running the implementation at edit time demands some form of sandboxing. For HYBRID CLOJURESCRIPT, the authors had to manually construct this sandboxing for the CodeMirror IDE using Stopify; if a team adapts another, more-powerful IDE, such as Visual Studio Code,[2] the existing language-server architecture may already account for this need.[3]

7 EVALUATION: PRESERVING AND ENHANCING A DEVELOPER'S WORKFLOW

Evaluating a language design should confirm that it is both *useful* and *usable*. Previous designs for interactive-syntax demonstrate its *usefulness* with a plethora of examples. Each validates that interactive-syntax expresses some domain concepts more directly and clearly than linear text. The next section sketches how such examples are easily reconstructable in the DOM-based approach.

The *usability* of previous designs, however, is questionable. Here, usability means that developers can build on what they know and can easily create and insert interactive-syntax extensions.

---

[2]https://code.visualstudio.com/
[3]https://microsoft.github.io/language-server-protocol/

More generally, a usable hybrid language should enhance—not interfere with—the ordinary software development workflow. This section presents a systematic characterization of major and minor workflow activities and an analysis of how well this paper's DOM-based design compares with prior designs to enhance and preserve them (section 7.1). Additionally, it addresses some remaining areas of Hybrid ClojureScript and elIDE that need improvement (section 7.2).

7.1 Workflow Operations and Interactive Syntax

Programmers interact with their codebases in the following major ways:

- *Auditing*, the most common task, is reading and comprehending existing code. The primary goal of visual and interactive syntax is to let code about geometric concepts speak for itself.
- *Creation*, the second-most common task, is to write new code. As far as interactive syntax is concerned, "creation" refers to two actions: (1) creating new interactive-syntax extensions and (2) using existing interactive-syntax extensions (from a library) to create programs. In the ideal case, a programmer working in a text-only IDE can still insert an instance of an interactive-syntax extension, and it must work correctly in a hybrid IDE.
- *Copy and Paste* is the act of copying code to, and pasting it from, the clipboard. It also refers to the direct action of dragging and dropping. Both are common, and interactive syntax must not get in the way of either.
- *Running* programs (in the IDE or otherwise) is a fundamental part of software development. Existing tools should work without changes, even if programs include instances of interactive-syntax extensions.
- *Search and Replace* is the act of finding code and, optionally, replacing it with new code. At a minimum, interactive syntax should not hinder these operations. Ideally, a developer should be able to search for graphical renderings of interactive syntax and/or replace existing code with graphical renderings of interactive syntax.

In addition to these five major actions on code, there is a significant number of more minor ones: *Abstraction*, *Autocomplete*, *Coaching*, *Code Folding*, *Comments*, *Comparison*, *Debugging*, *Dependency Update*, *Elimination*, *Hyperlinking Definitions and Uses*, *Merging*, *Migration*, *Multi-Cursor Editing*, *Refactoring*, *Reflow*, *Styling*, *Undo/Redo*. To avoid an overly long and tedious evaluation section, however, this section deals only with the major actions; a comparison of the minor actions with the most closely related approach can be found in the last section.

Figure 10 presents a comparison of the DOM-based design and other systems with respect to the major workflow operations. Specifically, it compares with Andersen et al. [2020], Livelits [Omar et al. 2021] and Sandblocks [Bein et al. 2020], which are the closely related projects with similar goals. Each cell in the two columns marks a workflow operation with a "✓" or a "✗", depending on how well the design works with this action.

As mentioned, the always-available text view is the key reason why all operations are possible with a DOM-based interactive-syntax approach. Every instance of an interactive-syntax extension is always available as both a graphical widget and a plain-text rendering because serialization works *inside* the IDE. Furthermore, all instances of interactive-syntax extensions are serialized to files as text, annotated with metadata. Hence every workflow operation can exploit the plain-text version, both inside and outside of the IDE.

The bespoke GUI library used in Andersen et al. [2020] is the key reason why two major workflow operations are difficult to impossible in that system. Specifically, the chosen IDE must internally store the bespoke GUI code as binary data—rendering existing workflow operations unavailable. For "creation," the bespoke GUI library does make it possible to create new types of

Activity	Literature	this paper	Andersen et al. [2020]	[Omar et al. 2021]	[Bein et al. 2020]
	System	—	Hybrid Racket	Livelits	Sandblocks
	Language	ClojureScript	Racket	bespoke language	Squeak
	GUI	DOM	bespoke GUI	DOM	Morphic
Auditing		✓	✓	✓	✗‡
Creation (definition)		✓*	✓*	✓*	✓*
Creation (use)		✓	✗†	✗†	✓
Copy and Paste		✓	✓	✗	✓
Running		✓	✓	✓	✓§
Search and Replace		✓	✗	✗	✓

    *   Orthogonal to interactive-syntax extensions.
    †   Possible, but difficult. See Section 2.
    ‡   Sandblocks limits interactive-syntax components to expressions.
    §   While possible, dynamic scoping leads to unexpected behavior. See Section 9.

Fig. 10. Interactive syntax vs coding actions

interactive-syntax extensions, but, using those new extension types is extremely challenging. Developers must frequently leave the IDE entirely to make relatively simple changes. Also, "Search and Replace" is limited to the functionality of the bespoke GUI library. As a result, developers must once again leave the IDE and use a plain-text editor.

Livelits uses a bespoke programming language. As such, the system fails to support acquired programming habits. It does, however, use the DOM GUI system, and it thus is conceivable that it could incorporate some ideas from this paper in order to improve some developer workflow operations. While both the DOM-based design of this paper and Sandblocks support all common editing operations, Sandblocks modifies Squeak's lexical scoping to dynamic scope. Further, interactive-syntax in this system may be created for expressions only. Thus, Sandblocks users are limited in which parts of a program may be expressed visually.

7.2 Minor Limitations

Hybrid ClojureScript, the hybrid language in this paper, is not perfect when it comes to usability. This subsection describes three shortcomings, so that potential hybrid language implementers are aware of them. None of these shortcomings are fundamental to interactive-syntax extensions, however, nor are they fundamental to a DOM-based design. Rather, they are limitations due to the chosen programming language, ClojureScript.

First, ClojureScript's macro system requires putting macro definitions and uses in separate files. Thus Hybrid ClojureScript introduces some workflow friction for programmers who wish to develop extensions and test instances in a single file. Fortunately, interactive-syntax extension definitions and uses can be placed in one file if they compile directly to a single run-time function.

Second, ClojureScript's macro system implements only a weak form of hygiene [Clinger and Rees 1991; Kohlbecker et al. 1986]. Thus, interactive-syntax elaborators are not hygienic. To circumvent this weakness, Hybrid ClojureScript provides a functional version of `elaborate` which suffices in most cases.

Finally, as briefly mentioned in section 6, Hybrid ClojureScript falls short in its sandboxing capabilities. While this has not posed any problem for any of the (almost 100) users of the prototype, it does highlight the need for future research on the interface between interactive syntax and

security. As is, the limited sandbox provided by Stopify means interactive-syntax extensions are, at worst, as secure as ordinary web pages designed with security in mind.

Although these limitations are undesirable, none of them reduce the usefulness or usability of Hybrid ClojureScript in a substantial way. In practice, the ability to use a rendering engine with multiple decades of engineering offsets the high friction of defining extensions and minor problems with hygiene and sandboxing.

8 EVALUATION: DOM-BASED INTERACTIVE SYNTAX AT WORK

A hybrid language based on a standard and widely used GUI library, i.e., a DOM-based one, comes with significant advantages. First, programmers have built many specialized libraries for multi-dimensional domains that can be used to implement interactive-syntax extensions. Second, using such libraries imposes almost no effort on a programmer. Third, if a library does not quite fit, it tends to be open source and thus easy to modify.

This section presents three uses of Hybrid ClojureScript and implicitly elIDE. The first one shows how to use a graph library as-is to express a REST API connection as a state-machine. The second illustrates the ease with which a slightly modified library can be used to express the geometry of a board game. The final re-creates the most sophisticated example of Andersen et al. [2020]'s work—a meta-syntax extension—with less code by reusing existing libraries.

8.1 Using and Combining Existing Libraries

A protocol for calling methods in a certain order, e.g., during authentication, is often expressed as a state-machine diagram. Thus it is a perfect use case for interactive-syntax. More specifically, to prevent misusing objects, library authors often inject run-time code which checks that methods are called in the right order. Without interactive-syntax, this code must be manually synced to the state-machine diagram present in the documentation. Of course, there is no guarantee that the code corresponds to the diagram. Worse, the diagram and the code are likely to get out of sync as the library evolves.

Using an interactive-syntax extension, a library author can describe a protocol graphically and have the corresponding run-time-checking code generated automatically. Consider an authentication protocol for a REST API on objects with three methods: auth, req, done. The protocol imposes the following constraints on these methods:

(1) use the auth method to send credentials and receive an authentication token in response;
(2) use the req method, with an endpoint URL and the valid authentication token, to repeatedly request data; and
(3) use the done method to end the authenticated session.

Figure 11 shows how a programmer may use an interactive-syntax extension to express the protocol as a state machine. This extension elaborates to a predicate which, given a sequence representing the history of method calls to an authentication object, determines whether it satisfies the protocol [Dimoulas et al. 2016].

The state machine consists of three states: start, good, and end. The shaded gray background of start shows that it is the starting state. Each state indicates, via the transitions emanating from it, the set of methods a client module can call. For example, in the good state, a client module can call either the req or done method. A transition is labeled with a method name plus predicates for the arguments and result. If the arguments and result satisfy the predicates specified on the transition, then the state machine moves to the next state. If no such transition exists, then the protocol is violated, and this violation is reported.

```
;; [Sequenceof Message] -> Boolean
;; Returns whether the sequence of messages satisfies the
;; authentication protocol.
(def satisfies-auth-protocol?
```
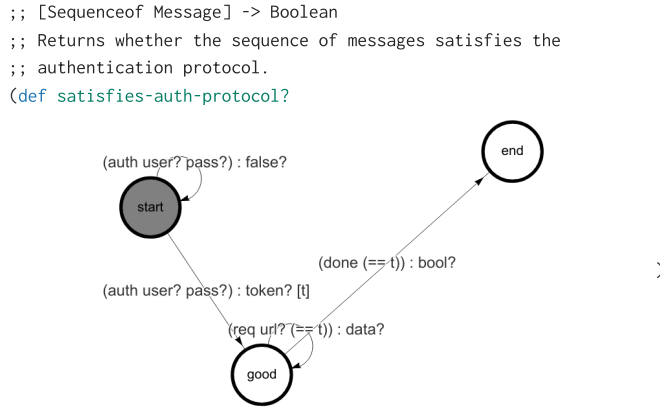

Fig. 11. A state machine for an API protocol

In figure 11 the transition corresponding to a successful authentication binds the returned token to the variable t. This is shown in square brackets. The scope of this binding includes all downstream transitions. Any transition in scope can then use this variable in predicates. For example, the expression (== t) constructs a predicate that determines if a value is equal to the token.

The diagram presented in figure 11 is actually just one use of a general-purpose interactive-syntax extension, which is used here to generate state-machine-checking predicates. To demonstrate the versatility of this extension, figure 12 shows a slightly simplified implementation of the Android MediaPlayer API[4] protocol that uses the same interactive-syntax extension.

```
;; [Sequenceof Message] -> Boolean
;; Returns whether the sequence of messages satisfies the
;; Android MediaPlayer protocol.
(def satisfies-android-protocol?
```
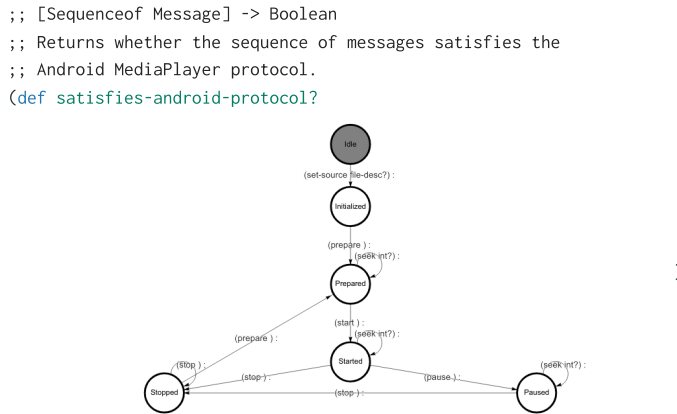

Fig. 12. The Android MediaPlayer protocol with interactive syntax

Using the interactive-syntax extension, a programmer performs GUI gestures to create new states; delete existing ones; add or delete transitions; edit the source and destination of a transition; turn states into starting or accepting states; rename states (via a text box); edit the predicates labeling a transition (via a text box); and change what variables are bound. These gestures are

---

[4]https://developer.android.com/reference/android/media/MediaPlayer

intuitive. For example, creating a new transition merely requires clicking and dragging from the source state to the destination state. Altering the properties of a transition involves selecting the transition and clicking the edit button.

The extension's elaborator analyzes code on the transitions to determine the necessary binding structure. Specifically, the elaborator creates a separate function for each transition with the appropriate parameters, and provides the run-time system enough information to supply the correct arguments to each function. Syntax and type errors in the specification are raised at compile time. For example, if a transition predicate specified a dependency on a variable that is not in scope, `elaborate` would signal a compile-time error.

Developing this kind of interactive-syntax extension is a relatively low-effort project. In a sense, it is a variation of the Bézier curve example as it uses the same generic graph-drawing component.[5] For the dialog to edit transition edges, the extension depends on a different GUI library.[6] As a result of this reuse-and-combine approach, the implementation for this extension consists of fewer than 300 lines of code.

8.2 Forking Libraries

Implementing a board game is another scenario where domain-specific geometric ideas dominate a number of activities. This subsection examines how the popular *Settlers of Catan* game can benefit from graphical syntax.

More specifically, implementing *Settlers* is challenging due to its hexagonal grid board where each edge of a hexagon is colored according to the player that owns that edge. A *road* consists of a continuous sequence of edges of the same color. When the game is scored, the longest such road plays a role.
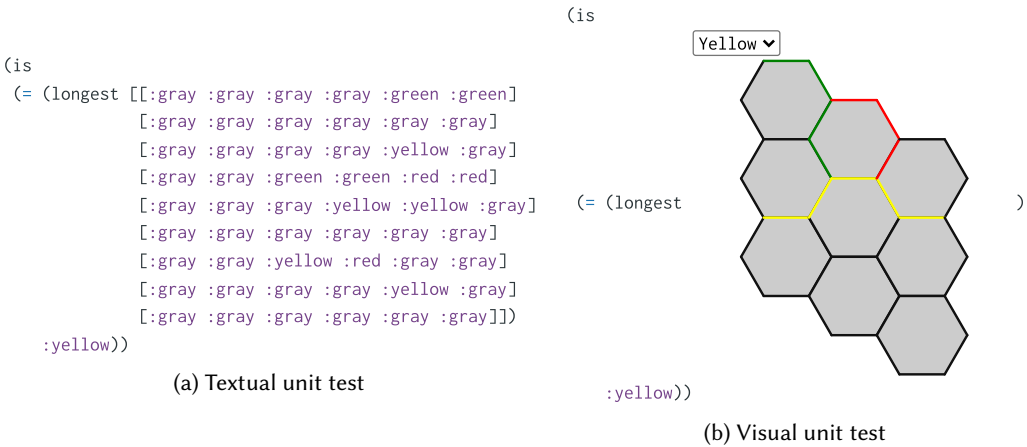

(a) Textual unit test

(b) Visual unit test

Fig. 13. An interactive-syntax extension for an implementation of "Settlers of Catan"

Unit tests demonstrate the usefulness of interactive-syntax extensions particularly well. In this spirit, figure 13a displays a unit test for the longest road calculation using traditional plain-text syntax. By contrast, figure 13b presents the same unit test using an instance of interactive-syntax extensions for tiles and boards. The board shows up exactly as it does in the application's GUI itself. Indeed, the interactive-syntax extension reuses GUI code from the application itself, making it

---

[5]https://visjs.org/
[6]https://getbootstrap.com/

simple to implement. If the GUI code of the game application were to change, the syntax extension would tag along. One consequence of this reuse concerns the manipulation of the board and tiles. To update the board, a programmer clicks directly on an edge to change its color. The desired color is selected via a drop-down menu (upper left).

It should be obvious from the two side-by-side figures that (1) text is an inferior medium to express and maintain the unit test and (2) an interactive-syntax representation comes with additional compile-time advantages, such as well-formed test inputs and outputs.

Most importantly, HYBRID CLOJURESCRIPT enables a programmer to implement the above scenario with about 50 lines of code and a small adaptation to an open-source library. Technically speaking, the code for the board uses a hexagon-grid library,[7] a mostly generic component for drawing hexagons. Unlike the libraries used in the preceding subsection, this library is not extensible. Thus, the authors had to fork it and add 45 lines of code in about two hours; these lines are generic, however, and would enhance the existing library for many other purposes.

8.3 Meta-Extensions

Meta-extensions are the most complicated form of interactive-syntax definitions. Roughly speaking, a meta-extension is an interactive-syntax extension whose instances elaborate to another interactive-syntax extension.

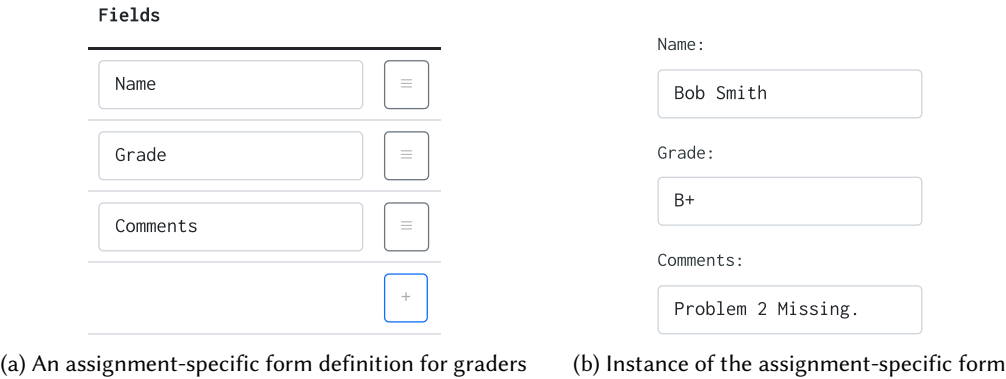(a) An assignment-specific form definition for graders   (b) Instance of the assignment-specific form

Fig. 14. An interactive-syntax extension for a form builder

This example concerns the editing of (tabular) forms, which are useful in the domain of software itself and many application domains. The case of editing forms is obviously self-referential, meaning an editor for a form must be able to generate forms. The concrete use case is about a programming instructor, who makes grading forms that teaching assistants can use to report a student's score.

Figure 14 illustrates how HYBRID CLOJURESCRIPT can realize such form editors. Specifically, Figure 14a displays a form editor for creating grading forms. In addition to creating new fields, the instructor can reorder fields and add optional constraints on the data stored in those fields. The `elaborate` computation of this interactive-syntax extension creates extensions whose instances look like the forms in figure 14b. Once filled with student-specific data, these generated forms elaborate to dictionaries, which can be submitted to the instructor's gradebook code.

---

[7]https://github.com/Hellenic/react-hexgrid

Both the form builder itself, as well as the forms created with that builder, exploit another DOM-based GUI library.[6] Using this library, the implementation is less than 50 lines of code. For comparison, Andersen et al. [2020], who created this example, report that their form builder code comes in at slightly more than 100 lines of code.

9   RELATED WORK

The first subsection compares with Sandblocks [Bein et al. 2020], a system with similar characteristics to the one presented in this paper. The remaining subsections look at works in other several areas that inspired this research: (1) languages and environments that allow programmers to run custom programs as they edit code; (2) graphical and non-textual programming languages; and (3) projectional and bidirectional editing.

9.1   Sandblocks

In February 2020, a research group at the Hasso-Plattner-Institut für Digitales Engineering at Universität Potsdam published a technical report [Bein et al. 2020] on the Sandblocks system. the Smalltalk programming language and its Morphic graphical development environment. At first glance, the visual syntax extensions look related to the ones presented here. Its visual extensions can be interleaved with program text. Furthermore, the project report carefully spells out the design goal that visual syntax must not interfere with a developer's tool chain and workflow.

Unfortunately, the design falls short of its goals. Unlike the hybrid language presented in this paper, visual elements in the Sandblocks implementation are not general. For example, programmers cannot add visualizations for field definitions, methods, patterns, templates, and other syntactic forms. Further, the visual constructs do not respect the language's static semantics such as lexical scope. As a result, the developer's toolchain and workflow are not preserved.

9.2   Edit Time

Two rather distinct pieces of work combine edit-time computation with a form of programming. The first is found 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 its Office productivity suite.

Spoofax [Kats and Visser 2010] is a framework for developing programming languages. Erdweg et al. [2011] recognize that, when developers grow programming languages, they would also like to grow their 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. The features are extra-linguistic, however, and thus do not support the kinds of abstraction (and meta-abstraction) enabled by interactive-syntax extensions.

Microsoft Office plugins, called VSTO Add-ins [Microsoft 2019], allow authors to create new types of documents and embed them into other documents. One developer might use it to make a music type-setting editor, while another might use it to put music notation in a PowerPoint presentation. Even though this tool set lives in the .NET framework, however, it is also an extra-linguistic idea and does not allow developers to build programming abstractions.

9.3   Graphical and Live Languages

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 board game tile, name it, and refer to this name in a unit test in a surrounding box. Boxer does *not*, however, satisfy any of the other desiderata listed in section 2. In particular, it has poor support for creating new abstractions with regard to the GUI elements.

Scratch [Resnick et al. 2009] is a fully graphical language system widely used 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, capabilities for a programmer to make new block types [Harvey and Mönig 2010]. These created block types, however, are themselves created through text.

LabVIEW [Kodosky 2020] is a commercial visual language targeted at scientists and engineers. It is widely adopted in its target communities. While it is possible to create robust products using LabVIEW, extending it with new types of visualizations is non-trivial, and it is rarely done.

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.

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

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 charts. Despite the apparent limitations of this domain, GRAIL was powerful enough to be implemented using itself.

Notebooks [Ashkenas 2019; Bernardin et al. 2012; Perez and Granger 2007; Wolfram 1988] and Webstrates [Klokmose et al. 2015; Rädle et al. 2017] are essentially a modern reincarnation of GRAIL, except that they use a read-eval-print loop approach to data manipulation rather than the GUI-based one made so attractive by the Morphic framework. These systems do not permit domain-specific syntax extensions.

9.4 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. The interactive-syntax mechanism in this paper is more general, however, and thus the authors of this paper conjecture that it could be used to implement a bidirectional editing system. Dually, ideas from other bidirectional editing systems could be used to improve the process of creating interactive-syntax extensions in the future.

Wizards and code completion tools, such as Graphite [Omar et al. 2012], perform 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.[8] Indeed, in this world, there are no programs per se, only graphically presented abstract syntax trees (AST) that 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.

Larch [French et al. 2014] also provides a hybrid visual-textual programming interface. Programs written in this 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. 2021] 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.

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 implies that programmers can write code with this plugin and view it in any plain-text editor. The plugin differs from interactive-syntax extensions, however, in two ways: (1) the plugins are less expressive than elaborators; and (2) the way new types of extensions are created limits programmers' ability to abstract over them. For example, programmers cannot create meta-instances with this plugin.

10 CONCLUSION

This paper describes a recipe for creating a hybrid programming language and IDE which can support the exact right mix of textual and interactive visual code that programmers need for their problem domain. Further, by starting with an appropriate existing language, IDE, and GUI library, the recipe produces hybrid results that are easy to use due to their familiarity to programmers; remain compatible with unadapted language implementations and tools; and also preserve a programmer's workflow. Finally, the paper demonstrates these benefits concretely by using the recipe to create Hybrid ClojureScript and a hybrid CodeMirror-based IDE. The evaluation shows that they improve on many of the shortcomings of prior hybrid textual-visual solutions.

The recipe discussed in this paper suggests several directions for future work. Here are some examples of possible future directions. First, there are several ways developers use visualizations when programming, everything from viewing charts to displaying call graphs. A future study can classify these visualizations and discuss how interactive syntax can handle them. Second, while this paper provides a recipe for turning a language into a hybrid variant, a language server protocol could be used to help automate this process. A future attempt can describe and analyze this protocol. Finally, while the application of the recipe to ClojureScript and CodeMirror is clearly successful, the next step is to demonstrate the applicability of the recipe to a language that is not already macro-extensible. The paper sketches how this can be accomplished; the only true proof, though, is an actual implementation.

---

[8]Intentional Software [Simonyi et al. 2006] seems related, but there is little information in the literature about this project.

ACKNOWLEDGMENTS

ADDITIONAL WORKFLOW EVALUATIONS

Section 7 evaluated the usability of various hybrid language systems with respect to the major operations that programmers use to edit software. This section focuses on the more minor programmer actions. Note that this evaluation considers a hybrid language usable even if it inhibits one of the minor coding actions. Nonetheless, an analysis of the minor actions is still included, with the understanding that each inhibited coding action increases the friction programmers experience when using interactive syntax and decreases its usability.

Activity	Literature System Language GUI	this paper — ClojureScript DOM	Andersen et al. [2020] Hybrid Racket Racket bespoke GUI
Abstraction		✗	✓
Autocomplete		✓	✓
Coaching		✓	✓
Code Folding		✗	✓
Comments		~	~
Comparison		✗	✓
Debugging		✗	✓
Dependency Update		✓	✓
Elimination		✗	✓
Hyperlinking Definitions and Uses		✗	✓
Merging		✗	✓
Migration		~	~
Multi-Cursor Editing		~	~
Refactoring		✗	✓
Reflow		✓	✓
Style		✓	✓
Undo/Redo		✓	✓

~   Orthogonal to interactive-syntax extensions.

Fig. 15. Interactive syntax vs coding actions, continued (also see 10)

The minor programmer actions that are evaluated are:

- *Abstraction* means generalizing two (or more) pieces of code into a single one that can then be instantiated to work in the original places (and more). Interactive syntax must facilitate converting one type of instance into another if abstraction involves the code for a definition of interactive syntax.
- *Autocomplete* allows programmers to choose descriptive names and enter them easily; recent forms of this code action complete entire phrases of code. It requires semantic knowledge of the programming language. In the ideal case, an IDE for a hybrid language should support autocompletion of textual prefixes into an instance of interactive syntax.
- *Coaching* is about the back-and-forth between programmers and analysis tools. A coaching tool displays the results of a (static or dynamic) analysis in the editor and (implicitly) requests a reaction. A simple example is the underlining of unbound variables; an advanced

one may highlight expressions that force register spilling. The challenge is that adding interactive syntax means extending the language in a non-functional manner, and doing so comes with its inherent problems.

- *Code Folding* enables IDEs to hide blocks of code while editing. Developers like to present overviews of code with code folded. Interactive syntax must not inhibit this IDE action.
- *Comments* do not affect interactive syntax.
- *Comparison*, often referred to as diffing, is needed to comprehend small changes to existing code as those are created. While the tool is available on say the text of git repositories, it is more often used inside of IDEs. If interactive syntax always comes with textual equivalents, code comparisons should continue to work in the conventional manner.
- *Debugging* demands running a program in a step-by-step fashion, i.e., steps a person can move through and comprehend sequentially. The comments of the preceding bullet apply here.
- *Dependency Update* is about updating packages and libraries for various reasons. A change made to the definition of an interactive-syntax extension is reflected in uses of that extension automatically.
- *Elimination* is the dual of abstraction, meaning in-lining the code for an existing abstraction. The comments concerning the abstraction bullet apply here, too.
- *Hyperlinking Definitions and Uses* allows programmers to easily navigate between definitions and uses. To hyperlink pieces of code properly, an IDE must understand both the text and the semantics of code. Whether this form of linking works properly depends on how easily an IDE can get hold of the text that corresponds to an instance of interactive syntax.
- *Merging* two blocks of code is the natural extension to comparison. Instead of viewing a report of the difference, however, merging attempts to generate syntactically correct code that represents two sources derived from one original point. Like comparison, merging is clearly a text-based action but more sophisticated. Research is needed to investigate how well merging works in the presence of interactive syntax.
- *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* allows two (or more) developers to concurrently edit the same program. This is orthogonal to interactive syntax.
- *Refactoring* is a syntax- or even semantics-aware search-and-replace action. Most simple refactoring actions should work as-is even in the presence of interactive syntax. More research is needed to understand whether refactoring works when syntactic differences involve interactive syntax.
- *Reflow* automatically transforms program text in an IDE buffer to conform to some style standards, e.g., proper indentation. If an IDE accommodates instances of interactive syntax, reflow continues to work.
- *Styling* changes aspects of code display, e.g., the font size or the color theme. Instances of interactive-syntax may benefit from explicitly coordinating with style operations.
- *Undo/Redo* is straightforward for text. For interactive syntax, each extension can package multiple changes into a single undo/redo step.

The table in figure 15 shows how Andersen et al. [2020]'s design compares with the one presented in this paper for each operation.

REFERENCES

Adobe. Adobe dreamweaver cc help, 2019. URL https://helpx.adobe.com/pdf/dreamweaver_reference.pdf.

Leif Andersen, Michael Ballantyne, and Matthias Felleisen. Adding interactive visual syntax to textual code. *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020. doi: 10.1145/3428290. URL https://doi.org/10.1145/3428290.

Jeremy Ashkenas. Observable: The user manual, 2019. URL https://observablehq.com/@observablehq/user-manual.

Samuel Baxter, Rachit Nigam, Joe Gibbs Politz, Shriram Krishnamurthi, and Arjun Guha. Putting in all the stops: Execution control for javascript. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, page 30–45, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356985. doi: 10.1145/3192366.3192370. URL https://doi.org/10.1145/3192366.3192370.

Leon Bein, Tom Braun, Björn Daase, Elina Emsbach, Robert Hirschfeld, Leon Matthes, Toni Mattis, Jens Mönig, Stefan Ramson, Patrick Rein, et al. *SandBlocks: Integration Visueller and Textueller Programmelemente in Live-Programmiersysteme*, volume 132. Universitätsverlag Potsdam, 2020.

Alexandre Bergel, Damien Cassou, Stéphane Ducasse, and Jannik Laval. *Deep Into Pharo*. Square Bracket Associates, 2013.

L. Bernardin, P. Chin, P. DeMarco, K. O. Geddes, D. E. G. Hare, K. M. Heal, G. Labahn, J. P. May, J. McCarron, M. B. Monagan, D. Ohashi, and S. M. Vorkoetter. Maple programming guide, 2012.

Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. Programmatic and direct manipulation, together at last. *SIGPLAN Not.*, 51(6):341–354, jun 2016. ISSN 0362-1340. doi: 10.1145/2980983.2908103. URL https://doi.org/10.1145/2980983.2908103.

William Clinger and Jonathan Rees. Macros that work. POPL '91, page 155–162, 1991. doi: 10.1145/99583.99607. URL https://doi.org/10.1145/99583.99607.

Christos Dimoulas, Max S. New, Robert Bruce Findler, and Matthias Felleisen. Oh lord, please don't let contracts be misunderstood (functional pearl). In Jacques Garrigue, Gabriele Keller, and Eijiro Sumii, editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 117–131, 2016. doi: 10.1145/2951913.2951930.

A. A diSessa and H. Abelson. Boxer: A reconstructible computational medium. *Commun. ACM*, 29(9):859–868, sep 1986. ISSN 0001-0782. doi: 10.1145/6592.6595. URL https://doi.org/10.1145/6592.6595.

Andrew D. Eisenberg and Gregor Kiczales. Expressive programs through presentation extension. In *Proceedings of the 6th International Conference on Aspect-Oriented Software Development*, AOSD '07, page 73–84, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 1595936157. doi: 10.1145/1218563.1218573. URL https://doi.org/10.1145/1218563.1218573.

Thomas O. Ellis, John F. Heafner, and W. L. Sibley. The grail language and operations. Technical Report RM-6001-ARPA, Rand. Corp. Santa Monica CA, 1969a.

Thomas O. Ellis, John F. Heafner, and W. L. Sibley. The grail project: An experiment in man-machine communications. Technical Report RM-5999-ARPA, Rand. Corp. Santa Monica CA, 1969b.

Sebastian Erdweg, Lennart C.L. Kats, Tillmann Rendel, Christian Kästner, Klaus Ostermann, and Eelco Visser. Growing a language environment with editor libraries. *SIGPLAN Not.*, 47(3):167–176, oct 2011. ISSN 0362-1340. doi: 10.1145/2189751.2047891. URL https://doi.org/10.1145/2189751.2047891.

Gerald Farin. *Curves and Surfaces for Computer-Aided Geometric Design: A Practical Guide*. Elsevier, 2014.

G. W. French, J. Richard Kennaway, and A. M. Day. Programs as visual, interactive documents. *Software: Practice and Experience*, 44(8):911–930, 2014.

Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman Publishing Co., Inc., USA, 1983. ISBN 0201113716.

D. Goodman. *The Complete Hypercard Handbook*. Bantam Books, Inc., USA, 1988. ISBN 055334577X.

Brian Harvey and Jens Mönig. Bringing "no ceiling" to scratch: Can one language serve kids and computer scientists? *Proc. Constructionism*, pages 1–10, 2010.

Brian Hempel, Justin Lubin, Grace Lu, and Ravi Chugh. Deuce: A lightweight user interface for structured editing. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 654–664, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356381. doi: 10.1145/3180155.3180165. URL https://doi.org/10.1145/3180155.3180165.

Daniel Ingalls, Krzysztof Palacz, Stephen Uhler, Antero Taivalsaari, and Tommi Mikkonen. The lively kernel a self-supporting system on a web page. In Robert Hirschfeld and Kim Rose, editors, *Self-Sustaining Systems*, pages 31–50, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-89275-5.

Lennart C.L. Kats and Eelco Visser. The spoofax language workbench: Rules for declarative specification of languages and ides. *SIGPLAN Not.*, 45(10):444–463, oct 2010. ISSN 0362-1340. doi: 10.1145/1932682.1869497. URL https://doi.org/10.1145/1932682.1869497.

Clemens N. Klokmose, James R. Eagan, Siemen Baader, Wendy Mackay, and Michel Beaudouin-Lafon. Webstrates: Shareable dynamic media. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software and Technology*,

UIST '15, page 280–290, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450337793. doi: 10.1145/2807442.2807446. URL https://doi.org/10.1145/2807442.2807446.

Amy J. Ko and Brad A. Myers. Barista: An implementation framework for enabling new tools, interaction techniques and views in code editors. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '06, page 387–396, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595933727. doi: 10.1145/1124772.1124831. URL https://doi.org/10.1145/1124772.1124831.

Jeffrey Kodosky. Labview. *Proc. ACM Program. Lang.*, 4(HOPL), jun 2020. doi: 10.1145/3386328. URL https://doi.org/10.1145/3386328.

Eugene E. Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce F. Duba. Hygienic macro expansion. LFP '86, pages 151–161, 1986. doi: 10.1145/319838.319859.

John Maloney and Walt Disney Imagineering. An introduction to morphic: The squeak user interface framework. *Squeak: OpenPersonal Computing and Multimedia*, 2001.

Microsoft. Office and sharepoint development in visual studio, 2019. URL https://docs.microsoft.com/en-us/visualstudio/vsto/office-and-sharepoint-development-in-visual-studio?view=vs-2017.

Cyrus Omar, Young Seok Yoon, Thomas D. LaToza, and Brad A. Myers. Active code completion. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 859–869, 2012. doi: 10.1109/ICSE.2012.6227133.

Cyrus Omar, David Moon, Andrew Blinn, Ian Voysey, Nick Collins, and Ravi Chugh. Filling typed holes with live GUIs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 511–525, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383912. doi: 10.1145/3453483.3454059. URL https://doi.org/10.1145/3453483.3454059.

Vaclav Pech, Alex Shatalin, and Markus Voelter. Jetbrains mps as a tool for extending java. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '13, page 165–168, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450321112. doi: 10.1145/2500828.2500846. URL https://doi.org/10.1145/2500828.2500846.

Fernando Perez and Brian E. Granger. IPython: A system for interactive scientific computing. *Computing in Science & Engineering*, 9(3):21–29, 2007. doi: 10.1109/MCSE.2007.53.

Bobby Powers, John Vilk, and Emery D. Berger. Browsix: Bridging the gap between unix and the browser. *SIGPLAN Not.*, 52(4):253–266, apr 2017. ISSN 0362-1340. doi: 10.1145/3093336.3037727. URL https://doi.org/10.1145/3093336.3037727.

Roman Rädle, Midas Nouwens, Kristian Antonsen, James R. Eagan, and Clemens N. Klokmose. Codestrates: Literate computing with webstrates. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, UIST '17, page 715–725, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349819. doi: 10.1145/3126594.3126642. URL https://doi.org/10.1145/3126594.3126642.

Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. Scratch: Programming for all. *Commun. ACM*, 52 (11):60–67, nov 2009. ISSN 0001-0782. doi: 10.1145/1592761.1592779. URL https://doi.org/10.1145/1592761.1592779.

Charles Simonyi, Magnus Christerson, and Shane Clifford. Intentional software. *SIGPLAN Not.*, 41(10):451–464, oct 2006. ISSN 0362-1340. doi: 10.1145/1167515.1167511. URL https://doi.org/10.1145/1167515.1167511.

Markus Voelter and Sascha Lisson. Supporting diverse notations in mps' projectional editor. In *GEMOC*, pages 7–16, 2014.

Markus Voelter, Daniel Ratiu, Bernhard Schaetz, and Bernd Kolb. mbeddr: An extensible c-based programming language and ide for embedded systems. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, page 121–140, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450315630. doi: 10.1145/2384716.2384767. URL https://doi.org/10.1145/2384716.2384767.

Stephen Wolfram. *The Mathematica Book*. Cambridge University Press, 1988.