Adding Interactive Visual Syntax to Textual Code

LEIF ANDERSEN, PLT and Northeastern University, United States of America
MICHAEL BALLANTYNE, PLT and Northeastern University, United States of America
MATTHIAS FELLEISEN, PLT and Northeastern University, United States of America

Many programming problems call for turning geometrical thoughts into code: tables, hierarchical structures, nests of objects, trees, forests, graphs, and so on. 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.

This paper proposes a novel mechanism for conveniently extending textual programming languages with problem-specific visual syntax. It argues the necessity of this language feature, demonstrates the feasibility with a robust prototype, and sketches a design plan for adapting the idea to other languages.

CCS Concepts: • **Software and its engineering** → **Visual languages**; **Macro languages**; **Domain specific languages**; *Graphical user interface languages*; **Integrated and visual development environments**; • **Human-centered computing** → *Human computer interaction (HCI)*; Accessibility technologies.

Additional Key Words and Phrases: Domain-Specific Language

ACM Reference Format:
Leif Andersen, Michael Ballantyne, and Matthias Felleisen. 2020. Adding Interactive Visual Syntax to Textual Code. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 222 (November 2020), 28 pages. https://doi.org/10.1145/3428290

1 TEXT IS NOT ENOUGH, PICTURES ARE TOO USEFUL

Code is a message from a developer in the present to a developer in the future, possibly the same person but aged. This future developer must comprehend the code and reconstruct the thoughts that went into it. Hence, writing code means articulating thoughts as precisely as possible.

Often these thoughts involve geometrical relationships: tables, nests of objects, graphs, etc. Furthermore, the geometry differs from problem domain to problem domain. To this day, though, programmers articulate their thoughts as linear text. Unsurprisingly, code maintainers have a hard time reconstructing geometrical thoughts from linear text, which reduces their productivity.

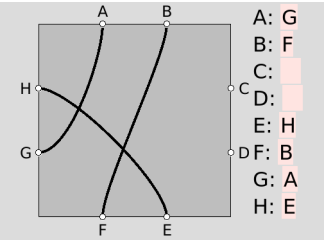
At first glance, visual languages [Boshernitsan and Downes 2004] eliminate this problem, but they actually don't. They too offer only a fixed set of constructs, though visual ones—meaning a visual language fails to address the problem-specific nature of geometric thought. And, as history shows, developers clearly prefer textual languages over visual ones—meaning graphical syntax should aim to *supplement*, not displace, textual syntax.

Authors' addresses: Leif Andersen, PLT, Khoury College of Computer Sciences, Northeastern University, 440 Huntington Ave., Boston, Massachusetts, 02115, United States of America, leif@leifandersen.net; Michael Ballantyne, PLT, Khoury College of Computer Sciences, Northeastern University, 440 Huntington Ave., Boston, Massachusetts, 02115, United States of America, mballantyne@ccs.neu.edu; Matthias Felleisen, PLT, Khoury College of Computer Sciences, Northeastern University, 440 Huntington Ave., Boston, Massachusetts, 02115, United States of America, mathias@ccs.neu.edu.

2475-1421/2020/11-ART222
https://doi.org/10.1145/3428290

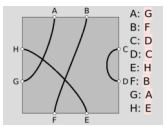
This paper presents a mechanism for extending textual languages with *interactive and visual* programming constructs tailored to specific problem domains. It demonstrates the feasibility of this idea with a prototype implementation. The design space description suggests the architecture could be adapted to a broad spectrum of programming languages.

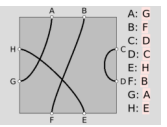
To make this idea precise, consider a software system that implements a game such as Tsuro.[1] In this game, players take turns growing a graph from square tiles, each of which displays four path segments. A player places one avatar at an entry point on the periphery of the grid-shaped board. New tiles 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 owner-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 a 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 simultaneously inserts it into the IDE. An instance is referred to as *editor*. Consider the image on the right, which shows a tile editor. It displays a graphical representation of the tile (left) and manual entry text fields (right). These text fields and graphical representation are linked. The graphic updates whenever a user updates the text; the text fields update when the programmer connects two nodes graphically via GUI 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"`.

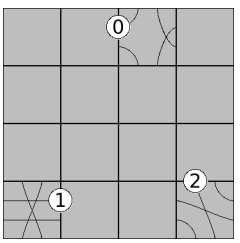
Every tile editor compiles to code that evaluates to a bidirectional hash-table representation of its connections. For the above example, the hash-table connects the `"A"` node with `"G"`, as the following lookup operations confirm:

```
> (hash-ref  'A)
'G
```

```
> (hash-ref  'G)
'A
```

Note how the editor itself assumes the role of a hash-table here.

A Tsuro developer will also create interactive syntax for Tsuro boards. 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 the nearby example. In this image, three players have clearly 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 a matrix of tiles.

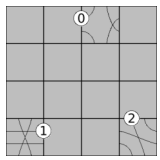
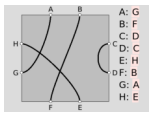
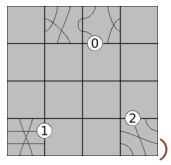


Once a programmer has extended the language with these two Tsuro-specific language constructs, a unit test using these graphical editors looks as follows in code:

[1] https://en.wikipedia.org/wiki/Tsuro

```
(check-equal?
  (send
    (new tsuro-board%
      [tiles (hash '(2 0) '(H D E B C G F A)
                   '(0 3) '(E F H G A B D C)
                   '(3 3) '(E C B H A G F D))]
      [players (hash player1 '(2 0 A)
                     player2 '(0 3 H)
                     player3 '(3 3 E))])
    addTile
    (new tsuro-tile% [connections '(G F D C H B A E)])
    player1)
  (new tsuro-board%
    [tiles (hash '(2 0) '(H D E B C G F A)
                 '(0 3) '(E F H G A B D C)
                 '(3 3) '(E C B H A G F D)
                 '(1 0) '(G F D C H B A E))]
    [players (hash player1 '(2 0 A)
                   player2 '(0 3 H)
                   player3 '(3 3 E))]))
```

Fig. 1. Textual Test Case

(check-equal? (send  addTile  player0) )

This one-line 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. The authors invite the reader to improve on this notational choice and compare their invention with the visual syntax above.

Interactive visual syntax is just syntax, and syntax composes. 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 the authors' 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—but, in the opinion of the authors, the message it sends to the future maintainer of the code is infinitely clearer than plain text could ever be.

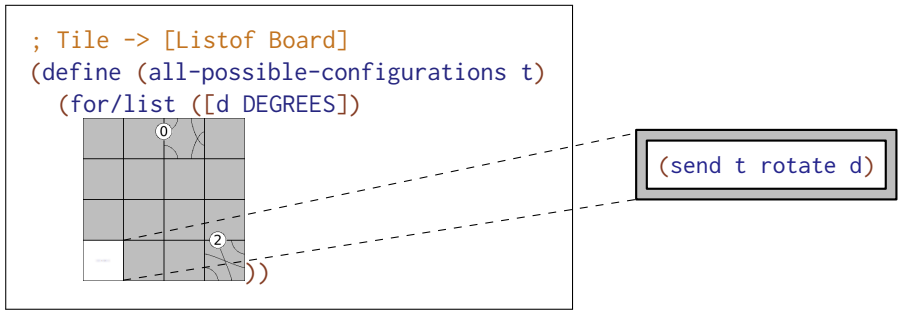


Fig. 2. Code Inside Interactive Visual Syntax

The preceding examples are code from the authors' code base, using a prototype implementation of the interactive-syntax extension mechanism. In fact, this paper itself is written using this prototype for live rendering and figure manipulation. While the prototype is implemented in Racket [Flatt and PLT 2010], the next section discusses a general design space, which should help language implementers to adapt this idea to their world. Following that, the paper demonstrates the expressive power of interactive syntax, explains the Racket implementation, contrasts it with other attempts at combining code with images, and concludes with a concise explanation of how the prototype falls short and what is needed to explore this idea from a user-facing perspective.

2 THE DESIGN SPACE

The project starts from an acknowledgment of the dominance of linear text. Its goal is to enable developers to supplement linear text with visual syntax as soon as they are tempted to document any code with some form of diagram. Furthermore, the transition from linear text to visual and interactive syntax must become as smooth as possible. Two prior projects [Eisenberg and Kiczales 2007; French et al. 2014] share the fundamental assumption and the ultimate goal of this project. Their shortcomings, however, expose the serious challenges of this undertaking.

In order to gain wide spread 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. Existing does not mean "widely used," though 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 usefulness as well as its usability.[2]

Second, more than one specific interactive development environment (IDE) must accommodate interactive syntax. Indeed, programs in the revised language must not even preclude editing in purely textual IDEs ones 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 a specific one.

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 interface of the software system, 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.

With these criteria in mind, we can now state specific design desiderata:

(1) *An interactive visual syntax is just syntax. It merely articulates an idea better than textual syntax.* If the underlying grammar distinguishes among definitions, expressions, patterns, and other syntactic categories, it should be possible to use visual syntax in all of them.

(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, say 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

[2]The popularity of notebooks, REPLs, 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. Techniques such as snapshots and cafés [Dybvig 2006] reduce the problem, but do not eliminate it.

of the edit-time code from other phases, say, the compilation phase and the runtime phase. This sandboxing greatly facilitates co-mingling code from different phases.

The Racket prototype realizes the design principles, and its use is illustrated next.

3 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 section describes interactive-syntax extensions, its parallels to traditional syntax extensions [Felleisen et al. 2018], and how these extensions can implement the Tsuro tiles from the previous section. The key novelty is `define-interactive-syntax`, a construct for creating new interactive-syntax forms.

3.1 Some Basic Background on Syntax Extensions

Racket comes with a highly expressive sub-language of macros that enable programmers to extend the language. 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

```
(define (f x)  _ _ _ elided _ _ _)
```

a programmer writes

```
(define-syntax (m x)  _ _ _ elided _ _ _)
```

to define the syntax transformer `m`. When the expander encounters `(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. The "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,

```
(begin-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 turned 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.

3.2 Editing As a Phase

Our technical idea is to support the *editing phase*, linguistically. The 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 module top level.

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 editor for this module is opened, or its content is modified. Like `begin-for-syntax`, `begin-for-interactive-syntax` 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:

interactive syntax module

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

`"Model/player.rkt"` module

```
(define (trace-player board player)
  _ _ _ elided _ _ _)
```

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

3.3 Bridging the Gap Between Edit-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`. While `define-syntax` allows run-time code and compile-time code to interact—the latter referencing and generating the former—`define-interactive-syntax` 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], the predominant Racket IDE, 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.

Concretely the `define-interactive-syntax` form consists of four sub-forms:

```
1 (define-interactive-syntax name$ base$
2   (define/public (draw ctx)
3     _ _ _ visually rendering _ _ _ )
4   (define/public (on-event event)
5     _ _ _ interactions code _ _ _ )
6   (define-elaborator
7     _ _ _ generating run-time code _ _ _ )
8   ( _ _ _  persistent data _ _ _ ))
```

The first line specifies the name of the interactive syntax and from which base class it is derived. 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, our 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. The extension is called `define-state` and its syntax is as follows:

```
1 (define-state name default
2    state-properties ...)
```

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 `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. However, 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 `define/public` are methods and capitalize on object inheritance. The `define-interactive-syntax` form ensures that `draw` and `on-event` are among the defined methods. The `define-state` form is mapped to the class fields. Finally, the `define-elaborator` form acts as a macro that generates run-time code. See section 5 for details.

3.4 Edit-Time Programming, a Tsuro Example

Implementing a dedicated state, renderer, event handler, and semantics for each interactive-syntax extension is somewhat labor intensive. To reduce the work load, our prototype implementation supports 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 `draw` and `on-event` methods. Inheritance works just like in Java. For example, every editor extends a `base$` class, which supplies basic drawing and event handling. Mixin functions abstract over inheritance. The `define-interactive-syntax-mixin` form creates new mixin types. These mixins are added to an editor's code by applying them to the editor's base class.

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 `vertical-block$` for vertical alignment, `horizontal-block$` for horizontal alignment, and `pasteboard$` for free-flowing editors. Each of these containers works with the `widget$` 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 `draw` and `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 `tsuro-picture$` editor (lines 18–20) works with `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 `text$$` and `focus$$` interactive-syntax mixins from the standard library; see figure 4. The `text$$` mixin (lines 1–5) handles both the text portion of the editor's state and drawing directly. The `focus$$` mixin (lines 7–10) handles user interaction. Finally, the `text-field$` editor (lines 12–14) combines the two mixins and applies them to the `widget$` base.

The code elaborator (lines 44–45) in figure 3 turns `pairs`, the state of the extension, into a hash table for the run-time phase, using the traditional syntax extension mechanism.

Importantly, developers may compose interactive-syntax extensions. Thus, for example, an instance of `tsuro-tile$` works with the interactive-syntax extension 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.

4 EVALUATING A PLETHORA OF EXAMPLES

The Tsuro-specific syntax extensions illustrate two aspects of programming with interactive visual 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." What might surprise readers (as it did the authors) is that there is barely any support for this idea in the world of programming languages. Perhaps language designers could not imagine how widely this idea is applicable or how to make this idea work easily.

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

```
(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 #`'#,(send this get-pairs)))
```

Fig. 3. Example Editor for Tsuro Tile

```
;; 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))
```

Fig. 4. Field editor using mixins

essential code of this particular example can be shared between the GUI interface to Tsuro and the unit test suites. A continued development of this prototype is likely to reduce the development burden even more, just like research on syntactic extensions has reduced the work of macro writers.

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 section starts with a list of inspirational sources: numerous text book illustrations of algorithms with diagrams, pictorial illustrations in standards such as RFCs, and ASCII diagrams in code repositories. The second subsection surveys a range of uses and our implementation of those uses, with an emphasis on where and how interactive syntax can be deployed. The final two subsections present two cases in some depth.

4.1 Examples of Diagram Documentation

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

Tree Algorithms. Every standard algorithms book and every tree automata monograph [Comon et al. 2007; Cormen et al. 2009] comes with many diagrams to describe tree manipulations. Programmers often include ASCII diagrams of trees in comments to document complex code.[3] These diagrams contain concrete trees and depict abstract tree transformations.

Matrix. Astute programmers format matrix-manipulation code to reflect literal matrices when possible.[4] Mathematical programming books depict matrices as rectangles in otherwise linear text [Fourer et al. 2002].

[3]https://git.musl-libc.org/cgit/musl/tree/src/search/tsearch.c?id=v1.1.21
[4]http://www.opengl-tutorial.org/

Name	Elements	LOC
Tree Algorithms	Data Literal, Pattern Matching, Templates	353
Matrix	Data Literal, Template	175
File System Data	Data Literal, Other Binding	178
TCP Parser	Data Literal, Pattern Matching, Templates	98
Pictures As Bindings	Other Binding	88
Video Editor	Data Literal, Template	80
Circuit Editor	Data Literal	307
Tsuro	Data Literal, Template	408
Form Builder	Data Literal, Template, Meta Binding	119

Fig. 5. Attributes of the Worked Language Extensions

File System Data Structures. Any systems course that covers the inode file representation describes it with box-and-pointer diagrams.[5] 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 are prone 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 SPICE [Vogt et al. 2019] simulation language 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.

4.2 The Expressive Power of Interactive-Syntax Extensions

We have implemented the examples from the previous section with interactive syntax. Doing so yields easily readable code and several insights on the expressive power of mixing visual and textual syntax. Here we 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.

[5] https://www.youtube.com/watch?v=tMVj22EWg6A

```
; Balance (- black) (= red)
;
;  /      -z-  |    -z-  |  -x-    |  -x-      \
; |       / \  |    / \  |  / \    |  / \       |        =y=
; |     =y=  D |  =x=  D | A  =z=  | A  =y=     |        / \
; |     / \    |  / \    |    / \  |    / \     | -->  -x- -z-
; |   =x=  C   | A  =y=  |  =y=  D |   B  =z=   |      /|   |\
; |   / \      |    / \  |  / \    |      / \   |     A B   C D
;  \ A   B     |   B   C | B  C    |     C   D /
(define/match (balance t)
  [(or (tree z 'black (tree y 'red (tree x 'red A B) C) D)
       (tree z 'black (tree x 'red A (tree y 'red B C)) D)
       (tree x 'black A (tree z 'red (tree y 'red B C) D))
       (tree x 'black A (tree y 'red B (tree z 'red C D))))
   (tree y 'red (tree x 'black A B) (tree z 'black C D))]
  [else t])
```

Fig. 6. A Textual Balance function for a Red-Black Tree

Data Literal. The simplest role is that of a data literal. In this role, developers interact with the syntax only to enter plain textual 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 by a lot of text—at the cost of reduced readability.

Template. The template's role generalizes data literals. Instead of entering plain data into an editor, 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. Languages such as Scala emphasize 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.

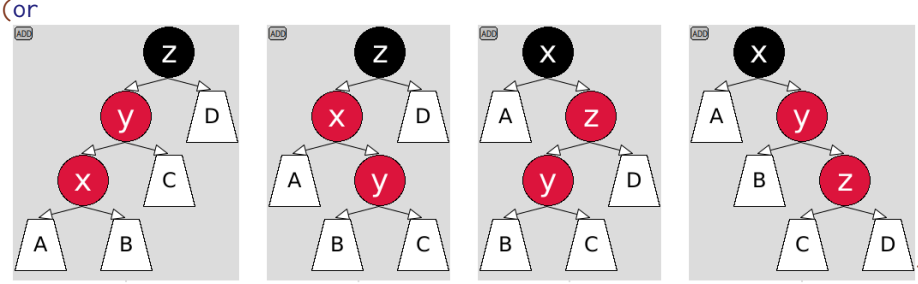
Meta Binding. Since syntax extension is a form of meta programming, interactive syntax naturally plays a meta-programming role, too. We refer to this role as meta-binding in figure 5. Here editors are used to construct new types of syntax, and because the prototype is in Racket, they can generate both graphical and textual syntax extensions. Section 4.4 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.

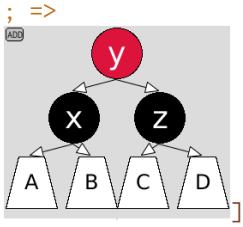
4.3 In Depth: Red-Black Trees

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

```
(define (balance t)
  (match t
    [(or
```




```
     ; =>
```




```
     ]
     [else t]))
```

Fig. 7. Visual Red-Black Tree Balance

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.[6]

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 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.

[6] https://blog.regehr.org/archives/1653

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

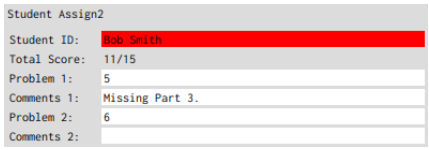
4.4 In Depth: Form Builders

The presented examples generate only run-time code, and they exclusively focus on patterns and data structures. Interactive-syntax extensions, however, can also generate compile-time and even edit-time code. This means that interactive-syntax 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.

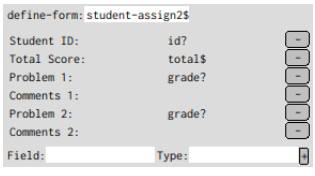
Programmers can add additional constraints to fields in those forms. Forms that do not meet these constraints are considered syntax errors. For example, here is a small program with an incorrectly filled grade form:




```
> (dict-ref                                    'score)
```

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 on the left 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$`, than the form uses it instead of the default text field. This enables custom sub-editors to display information

```
 1 (define-interactive-syntax form-builder$ vertical-block$
 2   ---elided---
 3   (define-elaborate this
 4     #:with name/sql              (retrieve-sql-name              this)
 5     #:with fields/sql            (retrieve-sql-field             this)
 6     #:with name$                 (retrieve-name                  this)
 7     #:with (field-name ...)      (retrieve-field-names           this)
 8     #:with (field-validator ...) (retrieve-field-validation-code this)
 9     ---elided---
10     #`(begin
11         (query-exec db (create-table name/sql
12                                      #:columns [fields/sql text] ...))
13         (define-interactive-syntax name$ table-base$
14           (new field$ ---elided---
15                [name field-name]
16                [on-change field-validator]) ...
17           ---elided---))))
```

Edit Time Compile Time Run Time

Fig. 8. Example Editor for a Form Builder

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%` 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 `define-interactive-syntax` 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, we 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.

5 THE IMPLEMENTATION OF AN INTERACTIVE-SYNTAX EXTENSION MECHANISM

The implementation of an interactive-syntax extension mechanism poses four challenges: editor syntax, edit-time semantics, editor elaboration, and IDE integration. This section first describes a general implementation approach that realizes our 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[7] including a connection to the DrRacket IDE. We conjecture that, with significantly more labor, this approach would also work for other languages with macro systems such as Clojure, Elixir, Julia, Rust, Scala, and even C++, as well as alternative IDEs.

5.1 Ingredients for an Interactive-Syntax System

As mentioned above, four efforts are required to implement support for interactive-syntax extensions to a programming language. First, the language's syntax must be augmented with a textual *editor* form to house 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 support interactive-syntax extensions.

The Editor Form. For a 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 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 parts: 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 elaborated 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 this. Languages that don't, such as Java, may still have tools that can emulate language extensions to a limited degree.

As an example, a form builder from the previous section may have this textual representation:

[7] https://github.com/videolang/interactive-syntax

```
$editor {
  binding: ["lib/form-builder.rkt", "form-builder$"],
  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 editor 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. If done externally, only one preprocessor is needed to support all interactive-syntax extensions.

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 section 3.

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 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, only one plugin is required to support all extensions.

Enabling an IDE to interpret editors demands two kinds of extensions. First, the IDE must recognize 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.

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 or a web browser 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.

5.2 A Prototype Using Racket and DrRacket

As a proof of concept, we have constructed a prototype of interactive-syntax extensions for the Racket language 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 like the form in the proceeding 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 we wish to demonstrate that interactive-syntax extensions can be added to a language without changing the underlying virtual machine or interpreter, the prototype employs a surprisingly robust work-around. We conjecture that such work-arounds exist for other languages too, though it is also likely that in many cases, an implementer would have to explicitly add 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 an example of such an elaboration. 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].[8] 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 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.

[8] Appendix A contains a brief introduction to submodules in Racket.

```
#lang editor-racket

(define-interactive-syntax simple$ base$
  (super-new)1
  (define-elaborate this
    #'(void)2))

#editor(#f . simple$)()

(begin-for-interactive-syntax
  (require editor/test)
  (test-window (new simple$)))
```

elaborates to

```
#lang racket

(provide simple$:elaborator)

(define-syntax (simple$:elaborator stx)
  (class/syntax base$:elaborator
                #'(void)2))

#editor(#f . simple$:elaborator)()

(module+ edit
  (provide simple$)

  (define simple$
    (class/interactive-syntax base$
                              (super-new)1))

  (require editor/test)
  (test-window (new simple$)))
```

Fig. 9. Elaboration of an Interactive-Syntax Extension

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.

```
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)]))
```

Fig. 10. Implementation of the `#%editor` macro

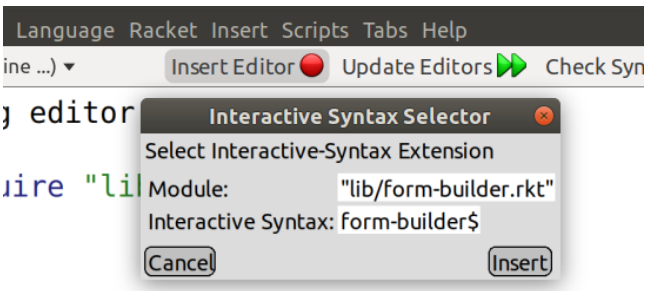
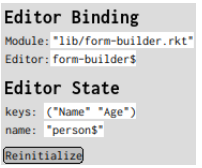


Fig. 11. Inserting an editor inside of DrRacket

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 them into specific points into 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 elements 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 shown on the right. Here the form builder 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 `Reinitialize` button tells the runtime system to make another attempt at re-initializing the editor with these values.

6 RELATED WORK

Three years ago at ICFP in Oxford, Andersen et al. [2017] presented the Video language. Video mixes text and editable video clips so that users can easily script productions, both textually and interactively. While this work on Video is a bespoke production, it raises the question of how to generalize this idea of graphical elements embedded in scripts and programs. Summarily speaking, this work empowers developers to

- create their own interactive-syntax extensions
- abstract with a plain syntax extension over interactive-syntax extensions
- abstract with an interactive-syntax extension over interactive-syntax extensions

In other words, interactive-syntax extensions are a proper part of the programming language in contrast to IDE-specific plugins, such as Video's.

In addition to drawing inspiration from the work of Andersen et al., this project also draws on ideas from research on edit time code, programming systems, and non-standard forms of editing.

6.1 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. Like Video, 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.

6.2 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 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 section 2. 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.

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 Smalltalk systems truly accommodate general-purpose graphical-oriented programming as a primary mode, however.

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.

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 this model, except that they use a read-eval-print loop approach to object manipulation rather than the GUI-based one, made so attractive by the Morphic framework. These systems do not permit domain-specific syntax extensions.

6.3 Projectional and Bidirectional Editing

Bidirectional editors attempt to present two editable views for a program that developers can manipulate in lockstep. Sketch-n-Sketch [Chugh et al. 2016; Hempel et al. 2018], for example, 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. We 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.[9] 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

[9]Intentional Software [Simonyi et al. 2006] has similar goals, but there is almost no concrete information in the literature about this project.

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 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, we conjecture that these constructs will not be deployed in the same range of linguistic contexts as interactive-syntax extensions (see section 4).

7 FROM LIMITATIONS TO FUTURE WORK

The prototype falls short of the plan laid out in section 2, though the shortcomings are of a non-essential technical nature, not principled ones:

(1) The prototype partially re-uses Racket's GUI library in the context of interactive-syntax definitions. At the moment, the prototype relies on a Racket-coded GUI tailored to the interactive-syntax system, meaning developers cannot use all GUIs for both syntax extensions and the application itself. The shortcoming is due to two intertwined technical reasons: the setup of Racket's GUI library and DrRacket's use of an editor canvas, which cannot embed controls and other window areas.

(2) The prototype does *not* validate the usability of the construction across different visual IDEs. It does accommodate the use of DrRacket and plain text editors such as Emacs or Vim. While the textual rendering of editors is syntactically constrained, a developer who prefers a text editor can still work on code and even read embedded interactive-syntax. Due to the design choice of relying on a Racket-based GUI for editors, though, interactive syntax will work in any IDE that can run Racket code and grant access to a drawing context such as a canvas. For details on how to engineer a general solution, see section 5.1.

(3) A minor shortcoming concerns editors that contain text fields into which developers enter code. In the current prototype, these text fields are just widgets that permit plain text editing. With some amount of labor, an interactive-syntax extension could use a miniature version of DrRacket so that developers would not just edit plain text, but code, in these places.

(4) Finally, the use of Racket's sub-modules to implement an edit-time phase falls short of the language's standard meta-programming ideals. While they *mostly* work correctly for mapping editors to code, the solution exhibits hygiene problems in some corner cases. Furthermore, while some meta-programming extensions in conventional languages, e.g., Rust, do implement hygienic expansion, others completely fail in this regard, e.g., Scala, which may cause additional problems in adapting this idea to different language contexts.

All of these limitations naturally point to future investigations.

Besides these technical investigations, the idea also demands a user-facing evaluation in addition to the expressiveness evaluation presented in section 4. Here are some questions for such a study:

- How quickly do developers identify situations where the use of interactive syntax might benefit their successors?
- How much more difficult is it to articulate code as interactive syntax than text?
- Is it easier to comprehend code formulated with interactive syntax instead of text?

Answers may simultaneously confirm the conjecture behind the design of interactive syntax and point to technical problems in existing systems. The authors will attempt to answer these question after hardening the prototype into an easily usable system.

8 CONCLUSION

Linear text is the most widely embraced means for writing down programs. But, we also know that in many contexts a picture is worth a thousand words. Developers know this, which is why ASCII diagrams accompany many programs in comments and why type-set documentation comes with elaborate diagrams and graphics. Developers and their support staff create these comments and documents because they accept the idea that code is a message to some future programmer who will have to understand and modify it.

If we wish to combine the productivity of text-oriented programming with the power of pictures, we must extend our textual programming languages with graphical syntax. A fixed set of graphical syntaxes or static images do not suffice, however. We must equip developers with the expressive power to create interactive graphics for the problems that they are working on and integrate these graphical pieces of program directly into the code. Concisely put, turning comments into executable code is the only way to keep comments in sync with code.

When a developer invests energy into interactive GUI code, this effort must pay off. Hence a developer should be able to exploit elements of the user-interface code in interactive-syntax extensions. Conversely, any investment into GUI elements for an interactive-syntax extension must carry over to the actual user-interface code for a software system.

Finally, good developers build reusable abstractions. In this spirit, an interactive-syntax extension mechanism must come with the power to abstract over interactive-syntax extensions with an interactive-syntax extension. If this is available to developers, they may soon offer complete libraries of interactive-syntax building blocks.

Our paper presents the design, implementation, and evaluation of the first interactive-syntax extensions mechanism that mostly satisfies all of these criteria. While the implementation is a prototype, it is robust enough to demonstrate the broad applicability of the idea with examples from algorithms, compilers, file systems, networking, as well as some narrow domains such as circuit simulation and game program development. In terms of linguistics, the prototype can already accommodate interactive syntax for visual data objects, complex patterns, sophisticated templates, and meta forms. We consider it a promising step towards a true synthesis of text and “moving” pictures.

ACKNOWLEDGMENTS

This research was partially supported by NSF grants 1823244 and 20050550. We also thank Nia Angle, Benjamin Chung, Benjamin Greenman, Elizabeth Grimm, Jason Hemann, Shriram Krishnamurthi, and Ming-Ho Yee for useful discussions and feedback on early drafts of this paper.

A SUBMODULES IN RACKET, A BRIEF REVIEW

Racket’s notion of submodule [Flatt 2013] is the key to the implementation of an edit phase. In Racket, submodules exist to organize a single file-size module into separate entities. A submodule does not get evaluated unless it is explicitly required, which is possible locally as well as from another file.

The introduction of submodules was motivated by two major desires: to designate some part of the code as `main` and to include exemplary unit tests right next to a function definition. Because submodules are separated from the surrounding module, adding such tests has no impact on the size or running time of the main module itself.

Consider this example:

```
;; inc : [Box Integer] -> Void
;; increment the content of the given box by 1

(module+ test
  (let ([x (box 0)])
    (check-equal? (unbox x) 0)
    (inc x)
    (check-equal? (unbox x) 1)))

(define (inc counter)
  (set-box! counter (add1 (unbox counter))))
```

The unit-test module creates a fresh counter box, initialized to `0`, increments it, and checks the value again. The programmer can evaluate these unit tests explicitly from another module or with a command-line tool:

```
[linux] $ raco test inc.rkt
raco test: (submod "inc.rkt" test)
2 tests passed
```

When `inc` is imported from another module, the unit tests are neither loaded nor run.

REFERENCES

Adobe. Adobe Dreamweaver CC Help. Retrieved May, 2020, 2019. https://helpx.adobe.com/pdf/dreamweaver_reference.pdf

Leif Andersen, Stephen Chang, and Matthias Felleisen. Super 8 Languages for Making Movies (Functional Pearl). *Proceedings of the ACM on Programming Languages* 1(International Conference on Functional Programming), pp. 30-1–30-29, 2017. https://doi.org/10.1145/3110274

Jeremy Ashkenas. Observable: The User Manual. Retrieved February, 2020, 2019. https://observablehq.com/@observablehq/user-manual

Rudolf Bayer. Symmetric Binary B-Trees: Data Structure and Maintenance Algorithms. *Acta Informatica* 1(4), pp. 290–306, 1972. https://doi.org/10.1007/BF00289509

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. Maplesoft, 2012.

Marat Boshernitsan and Michael S. Downes. Visual Programming Languages: a Survey. EECS Department, University of California, Berkeley, UCB/CSD-04-1368, 2004. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2004/6201.html

Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. Programmatic and Direct Manipulation, Together at Last. In *Proc. Programming Languages Design and Implementation*, pp. 341–354, 2016. https://doi.org/10.1145/2980983.2908103

Hubert Comon, Max Dauchet, Remi Gilleron, Florent Jacquemard, Denis Lugiez, Christof Löding, Sophie Tison, and Marc Tommasi. Tree Automata Techniques and Applications. 2007. http://tata.gforge.inria.fr/

Gregory Cooper and Shriram Krishnamurthi. Embedding Dynamic Dataflow in a Call-by-Value Language. In *Proc. European Symposium on Programming*, pp. 294–308, 2004. https://doi.org/10.1007/11693024_20

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. MIT Press, 2009.

Andrea A. diSessa and Harold Abelson. Boxer: A Reconstructible Computational Medium. *Communications of the ACM* 29(9), pp. 859–868, 1986. https://doi.org/10.1145/6592.6595

R. Kent Dybvig. The Development of Chez Scheme. In *Proc. International Conference on Functional Programming*, pp. 1–12, 2006. https://doi.org/10.1145/1160074.1159805

Andrew D. Eisenberg and Gregor Kiczales. Expressive Programs Through Presentation Extension. In *Proc. International Conference on Aspect-Oriented Software Development*, pp. 73–84, 2007. https://doi.org/10.1145/1218563.1218573

T. O. Ellis, J. F. Heafner, and W. L. Sibley. The GRAIL Language and Operations. RAND Corporation, RM-6001-ARPA, 1969a. https://doi.org/10.7249/RM6001

T. O. Ellis, J. F. Heafner, and W. L. Sibley. The Grail Project: An Experiment in Man-Machine Communications. RAND Corporation, RM-5999-ARPA, 1969b. https://www.rand.org/pubs/research_memoranda/RM5999.html

Sebastian Erdweg, Lennart C. L. Kats, Tillmann Rendel, Christian Kästner, Klaus Ostermann, and Eelco Visser. Growing a Language Environment with Editor Libraries. In *Proc. Generative Programming and Component Engineering*, pp. 167–176, 2011. https://doi.org/10.1145/2189751.2047891

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. A Programmable Programming Language. *Communications of the ACM* 61(3), pp. 62–71, 2018. https://doi.org/10.1145/3127323

Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: A Programming Environment for Scheme. *Journal of Functional Programming* 12(2), pp. 159–182, 2002. https://doi.org/10.1017/S0956796801004208

Robert Bruce Findler and PLT. DrRacket: Programming Environment. PLT Design Inc., PLT-TR-2010-2, 2010. https://racket-lang.org/tr2/

Matthew Flatt. Composable and Compilable Macros, You Want It When? In *Proc. International Conference on Functional Programming*, pp. 72–83, 2002.

Matthew Flatt. Submodules in Racket, You Want it When, Again? In *Proc. Generative Programming: Concepts & Experiences*, pp. 13–22, 2013. https://doi.org/10.1145/2517208.2517211

Matthew Flatt, Robert Bruce Findler, and John Clements. GUI: Racket Graphics Toolkit. PLT Design Inc., PLT-TR-2010-3, 2010. https://racket-lang.org/tr3/

Matthew Flatt, Robert Bruce Findler, and Matthias Felleisen. Scheme with Classes, Mixins, and Traits. In *Proc. Asian Symposium Programming Languages and Systems*, pp. 270–289, 2006.

Matthew Flatt and PLT. Reference: Racket. PLT Design Inc., PLT-TR-2010-1, 2010. https://racket-lang.org/tr1/

Robert Fourer, David M. Gay, and Brian W. Kernighan. AMPL: A Modeling Language for Mathematical Programming. 2nd edition. Cengage Learning, 2002. https://ampl.com/resources/the-ampl-book/

G. W. French, J. R. Kennaway, and A. M. Day. Programs as Visual, Interactive Documents. *Journal of Software: Practice and Experience* 44(8), pp. 911–930, 2014. https://doi.org/10.1002/spe.2182

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

Danny Goodman. The Complete Hypercard Handbook. Bantam Computer Books, 1988.

Brian Harvey and Jens Mönig. Bringing "No Ceiling" to Scratch: Can One Language Serve Kids and Computer Scientists? In *Proc. Constructionism*, pp. 1–10, 2010.

Brian Hempel, Justin Lubin, Grace Lu, and Ravi Chugh. Deuce: A Lightweight User Interface for Structured Editing. In *Proc. International Conference on Software Engineering*, pp. 654–664, 2018. 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 *Proc. Self-Sustaining Systems*, pp. 31–50, 2008. https://doi.org/10.1007/978-3-540-89275-5_2

Lennart C. L. Kats and Eelco Visser. The Spoofax Language Workbench. In *Proc. Object-Oriented Programming, Systems, Languages & Applications*, pp. 444–463, 2010. 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 *Proc. ACM Symposium on User Interface Software and Technology*, pp. 280–290, 2015. https://doi.org/10.1145/2807442.2807446

Amy Ko and Brad A. Myers. Barista: An Implementation Framework for Enabling New Tools, Interaction Techniques and Views in Code Editors. In *Proc. Conference on Human Factors in Computing Systems*, pp. 387–396, 2006. https://doi.org/10.1145/1124772.1124831

John Maloney, Kimberly M. Rose, and Walt Disney Imagineering. An Introduction to Morphic: The Squeak User Interface Framework. In *Squeak: Open Personal Computing and Multimedia*, pp. 39–77 Pearson, 2001.

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

Chris Okasaki. Red-black Trees in a Functional Setting. *Journal of Functional Programming* 9(4), pp. 471–477, 1999. https://doi.org/10.1017/S0956796899003494

Cyrus Omar, Nick Collins, David Moon, Ian Voysey, and Ravi Chugh. Livelits: Filling Typed Holes with Live GUIs (Extended Abstract). In *Proc. Workshop on Type-driven Development*, 2019.

Cyrus Omar, YoungSeok Yoon, Thomas D. LaToza, and Brad A. Myers. Active Code Completion. In *Proc. International Conference on Software Engineering*, pp. 859–869, 2012.

Mark Overmars. Teaching Computer Science Through Game Design. *Computer* 37(4), pp. 81–83, 2004. https://doi.org/10.1109/MC.2004.1297314

Vaclav Pech, Alex Shatalin, and Markus Voelter. JetBrains MPS as a Tool for Extending Java. In *Proc. Principles and Practice of Programming in Java*, pp. 165–168, 2013. https://doi.org/10.1145/2500828.2500846

Fernando Perez and Brian E. Granger. IPython: A System for Interactive Scientific Computing. *Computing in Science and Engineering* 9(3), pp. 21–29, 2007. https://doi.org/10.1109/MCSE.2007.53

Jon Postel. Transmission Control Protocol. Internet Engineering Task Force, RFC 793, 1981. https://tools.ietf.org/html/rfc793

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. *Communications of the ACM* 52(11), pp. 60–67, 2009. https://doi.org/10.1145/1592761.1592779

Roman Rädle, Midas Nouwens, Kristian Antonsen, James R. Eagan, and Clemens N. Klokmose. Codestrates: Literate Computing with Webstrates. In *Proc. ACM Symposium on User Interface Software and Technology*, pp. 715–725, 2017. https://doi.org/10.1145/3126594.3126642

Charles Simonyi, Magnus Christerson, and Shane Clifford. Intentional Software. *ACM SIGPLAN Notices* 41(10), pp. 451–464, 2006. https://doi.org/10.1145/1167515.1167511

Markus Voelter and Sascha Lisson. Supporting Diverse Notations in MPS' Projectional Editor. In *Proc. International Workshop on The Globalization of Modeling Languages*, 2014.

Markus Voelter, Daniel Ratiu, Bernhard Schaetz, and Bernd Kolb. mbeddr: an Extensible C-based Programming Language and IDE for Embedded Systems. In *Proc. Conference on Systems, Programming, and Applications: Software for Humanity*, pp. 121–140, 2012. https://doi.org/10.1145/2384716.2384767

Holger Vogt, Marcel Hendrix, and Paolo Nenzi. Ngspice Users Manual. NGSPICE, 30, 2019. http://ngspice.sourceforge.net/docs/ngspice-30-manual.pdf

Stephen Wolfram. The Mathematica Book. Fourth edition. Cambridge University Press, 1988.