

Adding Interactive Visual Syntax to Textual Code

ANONYMOUS AUTHOR(S)

Many programming problems call for coding geometrical thoughts: 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 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.

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.

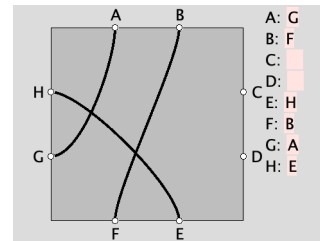
Often these thoughts involve geometrical relationships: tables, nests of objects, graphs, etc. Furthermore, the geometry differs from problem domain to problem domain. To this day, though, programmers articulate their thoughts as linear text. Unsurprisingly, code maintainers have a hard time reconstructing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

At first glance, visual languages [Boshernitsan and Downes 2004] eliminate this problem, but they 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.

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 Tsuru.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 Tsuru implementation. When a mechanism for creating interactive and visual syntax is available, the tester may add a Tsuru 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 imagine 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 users update 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".



¹<https://en.wikipedia.org/wiki/Tsuru>

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:

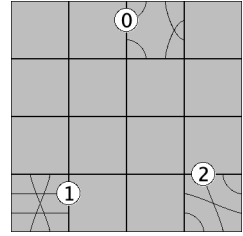
```
> (hash-ref (hash-ref 'G) 'A)
```



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

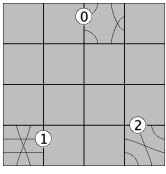
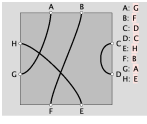
Note how the editor itself assumes the role of a hash-table here.

A Turo developer will also create interactive syntax for Turo boards. The board syntax supplies a grid of slots. Each slot is initially empty, but the programmer may place Turo tiles there to mimic players' moves. 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 cell in the top row), and each tile is occupied by an avatar. As far as run time is concerned, the Turo board editor evaluates to a list of lists of tiles.



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

```
(check-equal? (send (board) addTile (tile) (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 on the cell 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 choices 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 Turo 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.

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.

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

```

99      1 (check-equal?
100      2   (send
101      3     (new tsuro-board%
102      4       [tiles (hash '(2 0) '(H D E B C G F A)
103      5                        '(0 3) '(E F H G A B D C)
104      6                        '(3 3) '(E C B H A G F D))])
105      7       [players (hash player1 '(2 0 0)
106      8                        player2 '(0 3 7)
107      9                        player3 '(3 3 4))])
108      10    addTile
109      11    (new tsuro-tile% [connections '(6 5 3 2 7 1 0 4)])
110      12    player0)
111      13  (new tsuro-board%
112      14    [tiles (hash '(2 0) '(H D E B C G F A)
113      15                      '(0 3) '(E F H G A B D C)
114      16                      '(3 3) '(E C B H A G F D)
115      17                      '(1 0) '(G F D C H B A E))])
116      18    [players (hash player1 '(2 0 0)
117      19                      player2 '(0 3 7)
118      20                      player3 '(3 3 4))])])

```

Fig. 1. Textual Test Case

Once again, the developer can either express this context as a matrix 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 the message it sends to the future maintainer of the code is infinitely clearer than plain text could ever be.

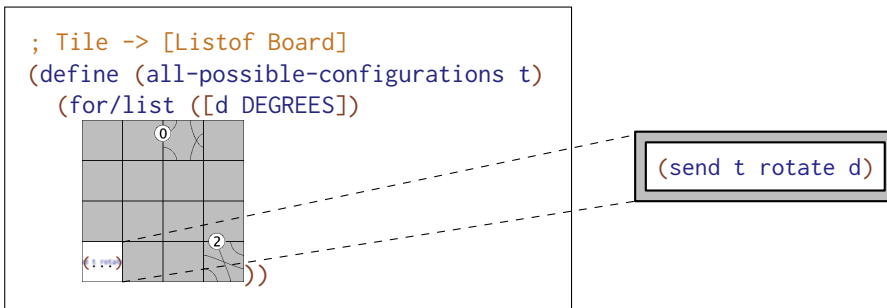


Fig. 2. Code Inside Interactive Visual Syntax

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, 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 interactive syntax from a user-facing perspective.

2 THE DESIGN SPACE

The ultimate goal of this project is to enable developers to extend their chosen language with interactive visual syntax the very moment they are tempted to document any code with some form of diagram. It acknowledges the dominance of linear text and demands that the transition from linear text to interactive syntax is smooth.

For the initial design, this goal implies two constraints. The design must first demonstrate the feasibility of mixed textual and interactive syntax in the context of an ordinary programming language. The design must further show how interactive visual syntax can be accommodated in more than one specific interactive development environment (IDE).

Additionally, 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 overarching considerations 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. A developer can then quit the IDE, and another can open this same file later, at which point the 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 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.* Furthermore, experience with syntax extension mechanisms suggests that it is also desirable to isolate the execution 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. 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 [Flatt and PLT 2010], and how these extensions can implement the Tsuru 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—which is why the macro expander takes a one-and-a-half pass approach 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. Naturally, function definitions for the compile-time phase may call macros defined for the compile-time phase of the compile-time phase. Programmatically a module may thus look as follows:

```

246 1 #lang racket
247 2 (define-syntax (m x) _ _ _ (f a b) _ _ _)
248 3 (begin-for-syntax
249 4   (define (f y z) _ _ _ (k c d e) _ _ _))
250 5   (define-syntax (k w) _ _ _ (g) _ _ _))
251 6   (begin-for-syntax
252 7     (define (g) _ _ _ elided))))

```

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

3.2 Editing As a Phase

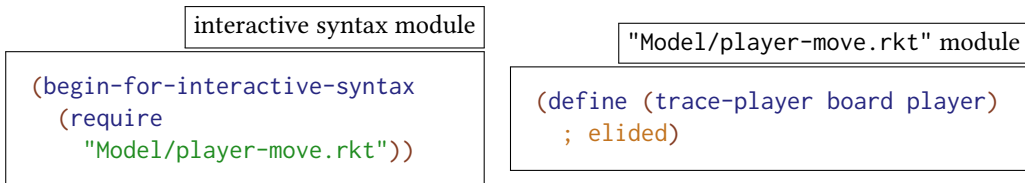
Our technical idea is to support the *editing phase*, in IDEs, linguistically. The extension to Racket consists of two linguistic constructs, analogous to `define-syntax` and `begin-for-syntax`:

- `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 Tsuru, for example, a `trace-player` function calculates the path of a player's token from the start position to the end position. It is used by the Tsuru-board syntax to calculate where it should draw the tokens after the placement of a tile, meaning the function is needed at edit-time:



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

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 consists of four different pieces:

- (1) a *presentation*, meaning a method for rendering its current state;
- (2) an *interaction*, that is, a method for reacting to direct manipulation actions;
- (3) a *semantics*, which is code that generates run-time code; and

(4) *persistent storage*, i.e., a specification of persisted data and its external representation.

Once an interactive-syntax definition exists, a developer may insert an instance into a DrRacket buffer by a UI action, such as a mouse click or button press. DrRacket 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.

The `define-interactive-syntax` form has four mandatory parts:

```
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 data 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   #:getter getter
3   #:setter setter
4   #:serialize serialize
5   #:deserialize deserialize
6   #:persistence persistence
7   #:init init
8   #:elaborator-default elaborator-default
9   #:elaborator elaborator)
```

Only `name` and `default` are mandatory fields; the remaining fields are optional and come with pragmatic default values. The `default` expression is evaluated during the first instantiation of the interactive syntax; its value persists across additional execution through automatic serialization. The optional function `serialize` marshals the data; the developer must specify this function if the persistent data cannot be automatically serialized. Likewise, `deserialize`, converts marshaled data back to its internal form. Both the `getter` and `setter` can be either a boolean or a function:

- `#f`, the default value, means that the state is inaccessible from the rest of the methods;
- `#t` translates to an unrestricted `getter` or `setter` method; and
- a function is useful to restrict access or mutation to the persistent state.

The `persistence` expression defines how long data should persist. For example, a document's text should persist across reloads of the document, but the cursor position should persist for only a short time. As with `getter` and `setter` the `persistence` expression can be either a boolean or a function. When provided, this function is run when the editor is saved; state values are not saved if this function returns `#f`. The `init` expression adds the state as an optional argument when an editor instance is created. Finally, the `elaborator-default` and `elaborator` fields act like `default` and `deserialize` respectively, but run during compile time rather than edit time.

Semantically an interactive-syntax definition creates a class. A definition using `define/public` or `define/private` turns into a method. Hence `define-interactive-syntax` ensures that `draw` and `on-event` are among the defined methods. The remaining pieces, `define-elaborate` and `define-state`, are mapped to combinations of private fields and methods. See section 5 for details.

3.4 Edit-Time Programming, a Tsuru Example

Implementing interactive-syntax extensions 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 predominate container blocks are `vertical-block$` for vertical alignment, `horizontal-block$` for horizontal alignment, and `pasteboard$` for free-flowing editors. Each of these containers work with the `widget$` editor type. Each child has a parent that supplies its drawing and event-handling methods.

For example, figure 3 presents the implementation of the Tsuru tile extension. The purpose of this interactive syntax is to permit the programmer to insert a graphical image of the tile where code is expected. The construction and maintenance of this tile demands a capability for connecting and re-connecting the entry points of the tile, for tracing the connections, and for displaying the current state of the connections both graphically and as text.

Rather than implementing the `draw` and `on-event` methods directly, the Tsuru syntax can use these container classes (lines 1, 15–34) to manage layout and events. These labels and fields (lines 18–38) are provided by the standard library and can draw themselves. The `tsuro-picture$` editor (line 16) 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:


```

393 1 (begin-for-interactive-syntax
394 2   (define TILE-NODES (list "A" "B" "C" "D" "E" "F" "G" "H")))
395 3
396 4 (define-interactive-syntax tsuro-tile$ horizontal-block$ (super-new)
397 5   ;; STATE
398 6   (define-state pairs (hash)
399 7     #:elaborator #t
400 8     #:getter #t)
401 9
402 10  ;; Char Char -> Void
403 11  ;; EFFECT connects letter to other and vice versa in pairs
404 12  (define/public (connect! letter other)
405 13    (send (hash-ref field-gui letter) set-text! other)
406 14    (send (hash-ref field-gui other) set-text! letter)
407 15    (set! pairs (hash-set* pairs letter other other letter))
408 16    (send picture set-tile! (draw-tile pairs)))
409 17
410 18  ;; VIEW : two horizontally aligned elements
411 19  (define picture (new tsuro-picture$ [parent this][record connect!]))
412 20
413 21  (define fields (new vertical-block$ [parent this]))
414 22  ;; Char -> Void
415 23  ;; EFFECT creates a text field as a child of fields
416 24  (define (add-tsuro-field! letter)
417 25    ;; TextField Event -> Void
418 26    ;; EFFECT connect the specified char in f with this letter
419 27    (define (letter-callback f e)
420 28      (connect! (send f get-text) letter))
421 29
422 30    ;; Container -> Void
423 31    ;; EFFECT create an option field as a child of p
424 32    (define (option-maker p)
425 33      (new field$ [parent p] [callback letter-callback]))
426 34
427 35    (new labeled-option$ [parent fields]
428 36      [label (format "~a: " letter)]
429 37      [option option-maker]))
430 38
431 39  (define field-gui ;; create all text entry fields
432 40    (for/hash ([a TILE-NODES])
433 41      (values a (send (add-tsuro-field! a) get-option))))
434 42
435 43  ;; CODE GENERATION
436 44  (define-elaborate this #'#, (send this get-pairs)))
437

```

Fig. 3. Example Editor for Tsuro Tile

```

442 1 ;; Mixin for drawing text in an editor
443 2 (define-interactive-syntax-mixin text$$
444 3   (super-new)
445 4   (define-state text "")
446 5   (define/augment (draw dc) ...))
447 6
448 7 ;; Mixin for basic user interaction
449 8 (define-interactive-syntax-mixin focus$$
450 9   (super-new)
451 10  (define/augment (on-event event) ...))
452 11
453 12 ;; A text field widget
454 13 (define-interactive-syntax text-field$ (focus$$ (text$$ widget$))
455 14   (super-new))

```

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$` editor (lines 12–15) combines the two mixins and applies them to the `widget$` base.

The code elaborator (lines 41–42) 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 Tsuru 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 A PLETHORA OF EXAMPLES

The Tsuru-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 1,000 words.” What might surprise readers (as it did the authors) is that support for this idea does not exist 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 essential code of this particular example can be shared between the GUI interface to Tsuru 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 (say, RFCs), and ASCII diagrams in code. The second section surveys a range of cases, with an emphasis on where and how interactive syntax can be deployed. The final two sections 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.² These diagrams contain concrete trees and depict abstract tree transformations.

Matrix. Astute programmers format matrix-manipulation code to reflect literal matrices when possible.³ This hand-crafting approach fails, however, for matrix transformation code.

File System Data Structures. Any systems course that covers the inode file representation describes it with box-and-pointer diagrams.⁴ 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. Code for parsing or generating such bit blocks benefits from diagram-based documentation.

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 their behavior refers to other actors using their 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, but engineers cannot mix and match textual and graphical part definitions.

4.2 The Expressive Power of Interactive-Syntax Extensions

Implementing all of these examples with interactive syntax 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 4 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.

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 code generator 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.

²<https://git.musl-libc.org/cgit/musl/tree/src/search/tsearch.c?id=v1.1.21>

³<http://www.opengl-tutorial.org/>

⁴<https://www.youtube.com/watch?v=tMVj22EWg6A>

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. 4. Attributes of the Worked Language Extensions

Template. The template’s role generalizes data literals. Instead of entering plain data into an editor, a developer adds code to these instances. The Tsuro board in the introduction and the tree in figure 6 are examples of such templates. The forms build a board and tree respectively using patterns 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 4. Here editors are used to construct new types of syntax, and because the prototype is in Racket, it 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, the programmer who must maintain 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.⁵

The balancing algorithm for red-black trees [Bayer 1972] illustrates this kind of work particularly well. Figure 5 shows a code snippet from a tree-manipulation library in Racket. The snippet depicts the left-oriented rotation of a binary tree. The comment block (lines 1–6) makes up the internal ASCII-art documentation of the functionality, while the code itself (lines 7–21) is written with Racket’s expressive pattern-matching construct.

An interactive-syntax extension empowers the developers to directly express the algorithm itself 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 both in the *pattern* part of a *match* expressions as well as in the *template* part, both situated within ordinary program text.

⁵<https://blog.regehr.org/archives/1653>

```

589 1 ; Rotate Left
590 2 ;      x          y
591 3 ;    / \      / \
592 4 ;  A  y  --> x  C
593 5 ;    / \      / \
594 6 ;   B  C      A  B
595 7 (define/match (rotate-left t)
596 8   [((struct* tree
597 9       ([value value] [left left] [color color]
598 10        [right
599 11         (struct* tree
600 12          ([value r-value]
601 13           [left r-left]
602 14           [right r-right]
603 15           [color r-color]))))]
604 16   (mk-tree #:left (mk-tree #:value value
605 17                   #:left left
606 18                   #:right r-left
607 19                   #:color color)
608 20   #:right r-right #:value r-value #:color r-color)])
609

```

Fig. 5. A Textual Left Rotate for a Binary Tree

A look at figure 6 makes this point for the red-black tree rotation 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 rotation 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 also clearly balanced relative to the matched subtrees.

In sum, the code mostly 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 interactive-syntax examples shown so far have only generated run-time code; the red-black tree Tsuru extension can only be used as patterns and data structures. Interactive-syntax extensions, however, can also generate compile-time and even edit-time code. In other words, editors can make new types of editors.

As an example, consider managing a large introductory course, with up to 800 students and a staff of 50 teaching assistants. The course coordinator will have to log the information for each student and staff member. Furthermore, each role requires different information. For example, each student gets a grade, while staff members do not. To manage all this information, the course runner

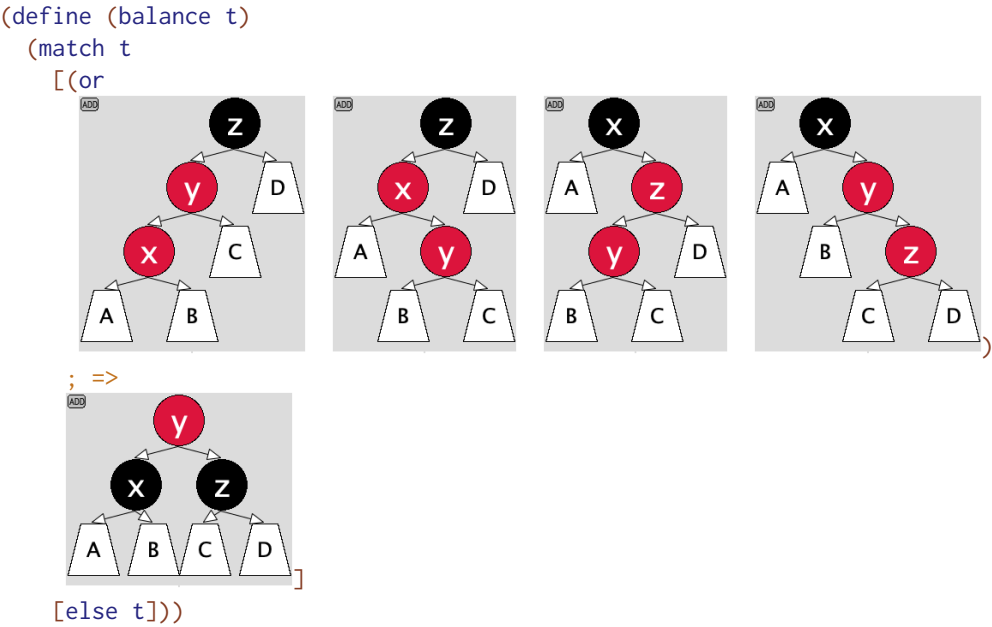


Fig. 6. Visual Red-Black Tree Balance

can use interactive syntax to create information forms. These forms enforce structured data, while visually indicating required information. Rather than making forms directly, the coordinator can create a form builder to generate each type of form.

To begin with, a course coordinator may create a course-specific form for recording student grades, hand lists of such forms to graders, and receive filled-in forms back, say one per assignment. Once the forms are returned, a script can access the fields of the forms:

Student Form

Student Name:

Bob Smith

Student ID:

12345

Student Email:

b.smith@university.edu

Grade:

B+

Comments:

Missing Problem 2.

```
(dict-ref grades)
```

define-form: student-form\$

Student Name:

-

Student ID:

-

Student Email:

-

Grade:

-

Comments:

-

+

Since the creation of visual forms is a common task and since this task is conceptually also visual, it is natural that the student forms are themselves created from a common interactive syntax extension for making forms. The editor on the left is an example of this meta-form. Using a direct click, the coordinator instantiated this form editor and gave a name to all its instances, via the text field labeled `define-form` at the top of the interactive syntax construct. As displayed, this meta-editor already specifies a list of fields: "Student Name", "Student ID", "Student Email", "Grade", and "Comment". Each field comes with a `-` button, with which the coordinator can remove the field from the form. At the bottom is a text field plus a `+` button; it permits the coordinator to add a labeled text field to `student-form$`. 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 "Phone Number" label, a correspondingly labeled, blank text field would show up in both of the editors in the list on the next line.

```

687 1 (define-interactive-syntax form-builder$ vertical-block$
688 2   (super-new)
689 3   (define-state name ""
690 4     #:setter #t
691 5     #:elaborator #t)
692 6   (define-state fields '())
693 7     #:elaborator #t)
694 8   ...
695 9   (define-elaborate this
696 10    #:with name$ (format-id this-syntax "~a$" (send this get-name))
697 11    #:with name-fields (format-id this-syntax "~a-fields"
698 12                        (send this get-name))
699 13    #:with fields (send this get-fields)
700 14    #`(begin
701 15      (begin-for-syntax
702 16        (define name-fields 'fields))
703 17      (define-interactive-syntax name$ table-base$
704 18        (super-new [keys 'fields]
705 19                    [name 'name$])
706 20        (define-elaborate this
707 21          #`'#,(send this get-table))))))

```

Fig. 7. Example Editor for a Form Builder







Likewise, a course coordinator may have forms to manage teaching assistants and graders. These forms may pair up graders to students. The nearby editors illustrate the forms an instructor would use. As with the student forms, the leftmost form is the template for the grader forms, with an actual instance to its right. Here, each grader has a name, an ID, an email address, and takes care of two students—fields that are created via the  button by filling out the text field at the bottom of the template form.

Figure 7 presents the implementation of the form builder syntax from section ???. It differs from the Tsuru tile implementation in two ways. First, it uses elements from the interactive-syntax standard library, which eliminates the need to create custom `draw` and `on-event` methods. Second, the elaborator creates a new editor type; that is, this interactive-syntax extensions is like a graphical GUI editor in that it generates another interactive-syntax extension.

The state of a form builder consists of the name of the new type of form and the names of its fields (lines 3–7). While the name is just settable, field names can be added and removed at will (lines 8–13). These `add-field!` and `remove-field!` methods are used by the view to update the state with the user's fields.

All that is needed for the view here is the `refresh-view` method (lines 14–16). Its purpose is to keep the view in sync with the current state. Hence both `add-field!` and `remove-field!` methods call it whenever a field is added or removed. The `vertical-block$` superclass also uses it when a refresh of the overall view is needed.

define-form: grader-form\$

Grader Name:	<input type="text"/>	
Grader ID:	<input type="text"/>	
Grader Email:	<input type="text"/>	
Student 1:	<input type="text"/>	
Student 2:	<input type="text"/>	
<input type="text"/>		

Grader Form

Grader Name:	Jane Allison
Grader ID:	31415
Grader Email:	
Student 1:	Bob Smith
Student 2:	Matt Fredrick

For this example, the elaborator is the key part (lines 17–28). Its purpose is to generate an interactive-syntax extension that consists of two parts. First, the elaborator defines the list of fields at edit time (line 24). Second it synthesizes the code for the form editor (lines 25–29).

The generated interactive-syntax extension looks deceptively simple:

- The new interactive-syntax class, named `name` as specified in the state of `form-builder$`, inherits from the `table-base$` superclass (lines 24). 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 is intentionally not shown here, because it is rather pedestrian GUI code.
- The elaborator of the generated interactive-syntax (line 28) simply expands to a hash table (dictionary) with the form’s field names and values.

In short, while generating interactive-syntax extensions from interactive-syntax extensions sounds complicated, decades-old syntax extension design patterns greatly facilitate the work.

In short, interactive visual syntax can be used to introduce new interactive visual syntax. Here a general form-generating interactive syntax construct is instantiated twice: once for creating student-specific forms and another time for creating grader-specific forms. In each case the instantiation adds another editor to the program. Additionally, the two meta-editors are instantiated from one common abstraction: an interactive visual language extensions for making forms. At some point, of course, the process has to switch back to text, and indeed, this is precisely the stage where it does.

5 THE IMPLEMENTATION OF AN INTERACTIVE-SYNTAX EXTENSION MECHANISM

The implementation of an interactive-syntax extension mechanism poses three open problems. The first problem concerns the semantic representation of editors, i.e. instances of interactive-syntax extensions. The second one is about getting IDEs to interpret editors. At a minimum, an IDE must permit the manipulation of editors as if they were text so that Emacs- and Vim-using developers can deal with them, too. At a maximum, an IDE must interpret editors, update their behavioral components when the “master” changes, and grant them access to its drawing-and-editing context. The third one is to accommodate the new editing phase within the language. While solving this last problem is somewhat idealistic, interactive syntax cannot safely be co-mingled with other forms of (syntactic) abstractions without separation of phases, and this new form of syntax should not create new security hole for IDEs.

This section first explains how these problems are solved for a Racket prototype (sections 5.1 through 5.3). The explanation suggests a plan on how to create similar implementations, possibly with weaker expressiveness and weaker guarantees, for other languages with macro systems such as Clojure, Elixir, Julia, Rust, Scala, and even C++ (section 5.4) and alternative IDEs.

A Short History of Graphical Syntax in the Racket World Over the past two decades, the Racket team has twice tried to add some form of geometrical syntax to its ecosystem.

The first approach uses a binary format, called WXME, to store modules that use non-textual syntax. Only a WXME-capable IDE can parse these binary files and insert interactive visual syntax as needed. As a result, these extensions belong to the IDE, not the language, making it impossible to create new ones within the language or abstract over them systematically. Also, building alternative WXME-enabled IDEs has proven difficult, due to a lack of a specification.

The second approach exploits unicode, including emojis, in text files. With unicode, it is possible, for example, to program interactive, two-dimensional conditionals.⁶ On the one hand, any contemporary IDE accommodates unicode editing. On the other hand, creating complex unicode characters and working with them is cumbersome at least and easily generates difficult-to-decipher syntax errors. Additionally, different GUI contexts display unicode differently, which may muddle the visual signal of the geometrical syntax.

5.1 The Editor Form

The Racket prototype interprets editors in analogy to closures. Like a closure, an editor combines a code pointer and its current state into a new kind of value. The code pointer 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 suffices to fully re-instantiate the editor as a graphical element in IDEs that can interpret these closures at edit time.

Concretely, an editor is represented in text as follows:

```
#editor(<binding>)(<state>)
```

The first section, `<binding>`, resembles a code pointer. It is a pair consisting of the relative file path to the defining module (`#false` for a locally defined one) and the name of the interactive-syntax extension in this file. The `<state>` is a hash table that maps each state variable to its serialization.

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.

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.

5.2 Cooperating With an IDE

Enabling an IDE to interpret editors demands two kinds of extensions. First, it must be possible to inform the IDE about interactive-syntax extensions so that it supports UI actions for the insertion of editors 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.

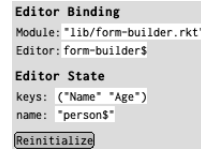
The prototype exploits DrRacket's plug-in API for the first concern [Findler and PLT 2010] and its Cairo-based drawing API for the second. A specially designed plug-in connects interactive-syntax extensions to the IDE. It inserts menu entries that programmers can use to instantiate interactive-syntax extensions and insert them into specific points into code. 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.

The prototype connects an editor to the IDE in three ways. First, it hands the editor a drawing context so that it can render itself within the IDE. Second, it forwards GUI events to the editor so that it can react to the programmer's UI actions. Third, it executes the editing code of an editor in a sandboxed environment so that the context is protected from malfunctioning (or malicious) GUI code in the editor.

⁶<https://docs.racket-lang.org/2d/index.html?q=2d>

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. Snips are extra-linguistic, however, that is, they are IDE-specific elements not elements of the language the programmer edits. The prototype bridges the gap between these two GUI elements so that editors remain language elements yet connect to the IDE smoothly.

Finally, the prototype accommodates failures with a simple fallback editor GUI. If one 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 just 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 try re-initializing the editor with these values.



5.3 Edit Phases

From a linguistic perspective, interactive-syntax extensions demands a edit phase, distinct from already existing compile-time and run-time phases. The addition of phase separation empowers developers to compose interactive-syntax extensions with syntactic abstractions and run-time abstractions. The language recognizes in which phase each element exists and acts accordingly. Enforcing the isolation of effects among these phases allows for clean, separate compilation of modules in the presence of co-mingled elements [Flatt 2002].⁷

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 implementor would have to explicitly add an edit phase.

Interactive-syntax extensions are implemented as syntactic 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 8 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]. Appendix A contains a brief introduction to submodules in Racket. This new, synthetic 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

⁷The paper explains the necessity of effect isolation for phases with six decades of experience with Lisp.

```

883 #lang editor-racket
884
885 (define-interactive-syntax simple$ base$
886   (super-new)1
887   (define-elaborate this
888     #'(void)2))
889
890 #editor(#f . simple$)()
891
892 (begin-for-interactive-syntax
893   (require editor/test)
894   (test-window (new simple$)))
895

```

elaborates to

```

899 #lang racket
900
901 (provide simple$:elaborator)
902
903 (define-syntax (simple$:elaborator stx)
904   (class/syntax base$:elaborator
905     #'(void)2))
906
907 #editor(#f . simple$:elaborator)()
908
909 (module+ edit
910   (provide simple$)
911
912   (define simple$
913     (class/interactive-syntax base$
914       (super-new)1))
915
916   (require editor/test)
917   (test-window (new simple$)))
918

```

Fig. 8. Elaboration of an Interactive-Syntax Extension

to the `simple$` interactive-syntax extension is refined to a reference to the elaborator; for edit time execution, the IDE plugin performs a separate name resolution. Finally, the test code in the `begin-for-interactive-syntax` block is also moved into the `edit` submodule.

Placing the edit-time code into a separate submodule permits the runtime system to distinguish between editor-specific code and general-purpose program code. In particular, it ensures that the runtime system can execute the editor portion of an interactive-syntax extension independently of its host module. Indeed, to do so, the runtime system merely requires the `edit` submodule and thus

obtains 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.

5.4 Other Programming Languages, Other IDEs

Experience with the prototype construction suggests basic guidelines for adapting the design plan to other contexts, both in terms of language and IDE.

As far as the linguistic side is concerned, an adaptation assumes the existence of a (n ideally hygienic) code-generating subsystem, such as the macro systems mentioned above, and demands the following changes to the language. First, the front end must deal with the equivalent of `#editor()`. The reader must accept this form and hand it to the parser, which, in turn, must generate a valid syntax tree. For languages with reader macros or similar extension mechanisms, this change can be implemented in a portable manner; for others, the addition of interactive syntax needs support from the implementor of the host language. Second, the compiler must translate definitions of interactive syntax and editors to valid code. If the assumed macro system is sufficiently expressive, this change does not require compiler surgery. Third, the ideal adaptation needs a proper implementation of phases, including effect isolation. Admittedly, though, the Lisp world has lived with macros and without phase separation for six decades, so perhaps a first-cut implementation may shift this burden to developers.

As far as the IDE side is concerned, an adaptation depends on the graphics tool kit for both Racket and other languages. The prototype benefits from a specific design choice, namely, a Racket implementation of a GUI toolbox tailored to the interactive-syntax extension mechanism. For other IDEs based on Cairo, an adaptation should be straightforward. For others, substantial engineering is required. The key idea is that any IDE can render interactive syntax and delegate events to editors if (1) it can run a Racket sub-process and (2) supplies a bridge object that mediates between its drawing context and the drawing context of the sub-process.

6 RELATED WORK

Two 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 begs 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 plain syntax extensions over interactive-syntax extensions
- abstract with an interactive-syntax extension over similar 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 Spoofox 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, Spoofox [Kats and Visser 2010]⁸ 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 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 extra-linguistic IDE plugins and thus do not enhance programming.

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 three most prominent examples are Boxer, Hypercard, 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 particular, it comes without any abstraction power with regard to the GUI elements: a user can collect primitive editors into a compound editor, but there is no mechanism to turn a compound editor into a primitive for new editors.

Scratch [Resnick et al. 2009], also an MIT product, pushes Boxer all the way to a 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 the together creates syntactically valid programs. Scratch offers limited capabilities for a programmer to make new block types.

Hypercard [Goodman 1988] is probably the oldest example of a graphical programming language. It gives users a graphical interface to make interactive documents. Authors have used hypercard to create everything from user interfaces to adventure games. Although 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; Klokmoose 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 text-oriented programming as a primary mode, however.

Notebooks [Ashkenas 2019; Bernardin et al. 2012; Perez and Granger 2007; Wolfram 1988] 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.

⁸Intentional Software [Simonyi et al. 2006] has similar goals, but there is almost no concrete information in the literature about this particular direction.

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. 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. 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.

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

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 constraint, 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 5.4.

- (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, say 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 contexts.

All of these limitations naturally point to future investigation.

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. We do *not* subscribe to the use of undergraduate students for such studies but intend to recruit Racket developers once the system is robust.

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. And 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 programmers 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.

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 even meta forms. We consider it a promising opening of a new path towards a true synthesis of text and “moving” pictures.

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. FrTime: Functional Reactive Programming in PLT Scheme. Brown University, CS-03-20, 2004. <https://cs.brown.edu/research/pubs/techreports/reports/CS-03-20.html>
- 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>
- 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>
- 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/>

- Adele Goldberg and David Robson. Smalltalk-80: The Language and Its Implementation. Addison-Wesley Longman Publishing Co, 1983. <https://dl.acm.org/citation.cfm?id=273>
- Danny Goodman. The Complete Hypercard Handbook. Bantam Computer Books, 1988.
- 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 Spoofox Language Workbench. In *Proc. Object-Oriented Programming, Systems, Languages & Applications*, pp. 444–463, 2010. <https://doi.org/10.1145/1932682.1869497>
- Clemens N. Klokmoose, 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>
- Andrew J. 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. <http://tydeworkshop.org/2019-abstracts/paper17.pdf>
- 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. <https://ieeexplore.ieee.org/document/6227133?tp=&arnumber=6227133>
- 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. Klokmoose. 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. <http://mbeddr.com/files/gemoc2014-MPSNotations.pdf>

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.

A A BRIEF INTRODUCTION TO SUBMODULES IN RACKET

Racket's notion of submodule [Flatt 2013] is the key to the implementation of an edit phase. The remainder of this subsection starts with a concise description of Racket's notion of submodule. It concludes by showing how to implement edit phase with submodules.

In Racket, submodules exist to organize a single file-size modules into separate entities. A submodule does not get evaluated unless it is specifically invoked. 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:

```
1 ;; inc : [Box Integer] -> Void
2 ;; increment the content of the given box by 1
3
4 (module+ test
5   (let ([x (box 0)])
6     (check-equal? (unbox x) 0)
7     (inc x)
8     (check-equal? (unbox x) 1)))
9
10 (define (inc counter)
11   (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 called from another module, the unit tests are neither loaded nor run.