Abstract

We investigate the apparent failure of visual programming to gain mainstream adoption and introduce the notion of General Visual Programming (GVP) as a framework for analyzing such systems. We review several existing approaches in light of this definition and identify which of them can be considered GVP systems, discussing their limitations that go beyond visual representation itself. We also consider some desirable features of programming environments that are not necessarily related to GVP. Finally, we present the design and implementation of GRASP, our experimental GVP system, highlighting major design decisions, lessons learned, and future directions.

1 Introduction

Visual programming systems appear to improve the efficiency of certain aspects of programming, compared to their textual counterparts. This advantage is followed by deserved popularity in certain niches - for example, the makers of the educational visual programming environment *Scratch* (Resnick et al. 2009) claim to have over 135 million registered users as of 2024 (Foundation 2024).

The success of visual programming environments isn't limited to education, though. Some other important domains, such as dynamic system modeling, game scripting or instrument control and industrial automation, are also covered by visual programming tools like Simulink (*Simulink Documentation* 2025), Unreal Engine Blueprint (*Unreal Engine: Blueprint Visual Scripting* 2025) or LabVIEW (*LabVIEW Documentation* 2025).

However, when we look at programming language popularity rankings, it turns out that they are firmly dominated by programming languages that are based on text. According to the programming language popularity ranking maintained by TIOBE as of 2025, the most popular visual programming language included in the ranking is Scratch, whose peak popularity between 2009 and 2025 was around 1.8% (by comparison, the peak popularity of the highest-ranked language, Python, is over 25%). And although the methodology employed by TIOBE is not infallible, it seems to reflect the everyday experiences of programmers: that in the professional setup, the dominance of textual programming languages is unquestionable (*TIOBE Index* 2025).

It is not hard to see why this is the case: text is, overall, a very elastic medium to impose strict meaning on, and the digital infrastructure – from teletypes to e-mail communication to copy-paste clipboard operations – is very well adapted to working with textual data, and the ubiquitous ASCII encoding maps very nicely to computer memory.

On the other hand, even the most popular visual programming systems seem to struggle to escape from their usually shallow niches to the realm of "general programming, where everything is possible". It is rather uncommon for such systems to be used for the development of themselves (while using text-based tools, such as Emacs or Eclipse, for their own development, is completely unsurprising, and being *self-hosted* is a very desirable property for compilers).

Moreover, if we take a closer look at e.g. the way Scratch stores its programs, it turns out that underneath they are just ZIP archives containing a bunch of files storing objects serialized to the textual JSON format.

2 General Visual Programming

Perhaps the fundamental problem with the framing of visual programming systems is that researchers often try to draw a sharp boundary between them and textual programming languages (even despite the fact that text itself is a visual medium), rather than trying to see the continuity between them. Even programs written in the most classic text-based environments can be presented and explored visually as icons embedded in a hierarchical structure, using tools such as Windows Explorer, Apple Finder or a programming environment's file browser - only such visualizations are typically shallow and do not expose the underlying semantic structure of programs.

Visual programming systems have proven their worth only in very specific and very limited domains. In this regard, they resemble *Domain-Specific Languages* (DSLs), which are specialized languages of limited applicability, suitable for describing problems that appear in particular subject domains.

The approach to system construction based on designing DSLs is sometimes called *Language-Oriented Programming*.

Probably the most striking example of Language-Oriented Programming is the Racket programming system (Felleisen, Findler, et al. 2018), developed by the PLT group, although even the UNIX operating system, whose components are known to employ parser generators such as TMG or YACC, is also sometimes considered a Language-Oriented Programming environment (Kernighan 2019).

In this paper, we propose the notion of *General Visual Programming*, which is an extension to Language-Oriented Programming that allows to define arbitrary Domain-Specific Notations/ways of interaction for programmers to work with their code.

General Visual Programming requires a specialized editor in order to be carried out. The editor needs to have the following properties:

- it needs to provide a base system, in which programmers can define specialized editors
- it needs to be able to embed the specialized editors within the base system
- the state of the specialized editors needs to be persisted along with the representation of code expressed in the base system

In principle, we cannot exclude the possibility of programmers designing their own Domain-Specific Notations, whose domain would be the design of particular Domain-Specific Notations. On the contrary, we consider this endeavour to be an interesting research direction in its own right.

The generality of General Visual Programming does not mean that the editor should be able to support every existing programming language. It only means

that it should support at least one general-purpose programming language as its base system.

3 Evaluating existing General Visual Programming systems

A number of existing programming systems can be analyzed through the lens of General Visual Programming.

3.1 Dr Racket

Dr Racket is an Integrated Development Environment designed for the Racket programming system with programming education in mind (Findler and PLT 2010). The Racket itself is derived from the Scheme programming language, so it inherits its *heavily-parenthesized* syntax of *s-expressions*. However, its `#lang` extension mechanism allows to support languages with arbitrary syntaxes.

In addition to being an academic research vehicle, it comes with a rich educational curriculum.

The IDE is implemented in the Racket programming language, which extends the Scheme language with its idiosyncratic Object-Oriented Programming system (Flatt, Findler, and Felleisen 2006).

The Dr Racket environment is focused mainly on textual languages, although it is quite unique in that it allows to use images (inserted to the editor via system clipboard or file browser). However, inserting an image to the program source makes the whole program incomprehensible to people who use plain text editors.

The images (called *snips*) can be assigned to variables and passed to functions. They can also be used to create other images via means of composition provided by the language.

The Racket programming environment provides a very interesting purely functional model of creating interactive applications called *Universe* (Felleisen and Team 2025). Racket has also been used as the core of an artistic live-programming environment called *Fluxus: Livecoding and Rapid-Prototyping Environment* n.d.

Despite very good support for graphics, and being an explicitly language-oriented programming system, Dr Racket alone cannot be considered a General Visual Programming system. However, the design of the environment is fundamentally open to visual extensions, and there is a line of research from the Racket community which tries to adapt it as such.

It is going to be described in the next section.

3.2 Interactive Visual Syntax

The paper *Adding Interactive Syntax to Textual Code* (Andersen, Ballantyne, and Felleisen 2020) describes a principled attempt to turn dr Racket into a

General Visual Programming environment. It dwells on the notion of code execution phases that were specified for the Racket programming language (which allows for arbitrary code execution during compilation time for the purpose of macro expansion) – so for example, there is a syntax expansion phase, as well as runtime execution phase.

The *Interactive Syntax* project adds another conceptual phase, namely *editing* phase.

It provides special forms analogous to `define-syntax` and `begin-for-syntax` (which are used for writing code that runs at macro-expansion time), namely `define-interactive-syntax` and `begin-for-interactive-syntax`, which are evaluated at edit time.

The `define-interactive-syntax` form requires to define the following methods:

- `draw`, which specifies how the visual interactive extension should be rendered
- `on-event`, which specifies how the visual interactive extension should react to user input
- an *elaborator*, which specifies how the visual extension should be converted back to the textual form
- code that handles *persistent storage* of the visualization (the motivational example provided in the paper is that "user may expect the text [of some particular visual extension] 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".).

The implementation of the `define-interactiv-syntax` form leans on Racket's Object-Orientation system.

Taken as a whole, the language extension also adds two buttons to the Dr Racket IDE - *Insert Editor* and *Update Editors*. The latter forces the program text to be re-scanned in search of the occurrences of the `#editor` special form.

Unfortunately, even though using interactive syntax doesn't spoil the textuality of the source code the same way that using snips with images does, the `#editor` special form is mostly illegible to humans.

The *Interactive Syntax* project has been further elaborated in Leif Andersen's PhD thesis, *Adding Visual and Interactive-Syntax to Textual Programs* (Andersen 2022). In the course of her work, she concluded that the Racket's GUI library (used, among other things, to build the Dr Racket IDE) isn't flexible enough to be reused inside the domain-specific editors.

For this reason, Andersen decided to build a browser-based IDE that would implement interactive visual syntax for ClojureScript, where the common GUI elements could be reused between the visual extensions and the application being developed.

This new approach also improved the textual representation of domain-specific editors, and the visual interface allows to present the textual variant of code alongside the visual one.

The treatment of visual extensions as "syntax", in the tradition of Lisp-based systems, is probably the most general approach to General Visual Programming conceivable: it is imaginable to not only use domain-specific editors for representing data structures, but also the code itself: it's possible to imagine control-flow or data-flow editors that would expand directly to code that implements the same logic.

Even though the examples presented in Andersen's work do not reflect this generality entirely, they exemplify the following roles of interactive syntax:

- data literals
- templates (from which literals can be constructed)
- algebraic matcher's patterns
- binding forms

The Racket-based realization of interactive syntax presented an implementation of Tsuro board game, Okasaki's algorithm of Red-Black Tree balancing, Form Builders and a few other *worked examples* (most of which are available in the evaluation artifacts repository[1]).

In particular, the Red-Black Tree example is notable for using visual extensions not only in regular code position, but also as patterns inside a pattern matching form, and templates for constructing a tree (cf. figure 1).

The ClojureScript realization of interactive syntax goes even further in this regard – it provides a new special form to the language called `g/let`, that can be used with a particular type of visual extensions that are desugared to an expression that can be placed inside of ClojureScript's core binding form, so that the components of the visual extension can serve to name certain components of some structure that is represented by that extension.

The exact definition of the `g/let` form in ClojureScript is the following (Andersen, Moy, et al. 2024):

```
(defmacro (g/let [diagram] & body)
  `(clojure.core/let ~(macroexpand diagram) ~@body))
```

The ClojureScript realization of visual interactive syntax also includes the Tsuro example, Form Builders, Bézier curve mid-point computation, a hexagonal board for the game Settlers of Catan (serving as a demonstration of the synergy that interactive syntax can get from running in the web browser environment, as it is based on a third-party JavaScript library) and a few others.

Andersen also build a "bridge" to be able to execute Racket code from within the browser by connecting the browser-based IDE to a language server via websockets. While she called this solution "a Frankenstein monster", she admitted

[1] `https://github.com/LeifAndersen/artifact2020`

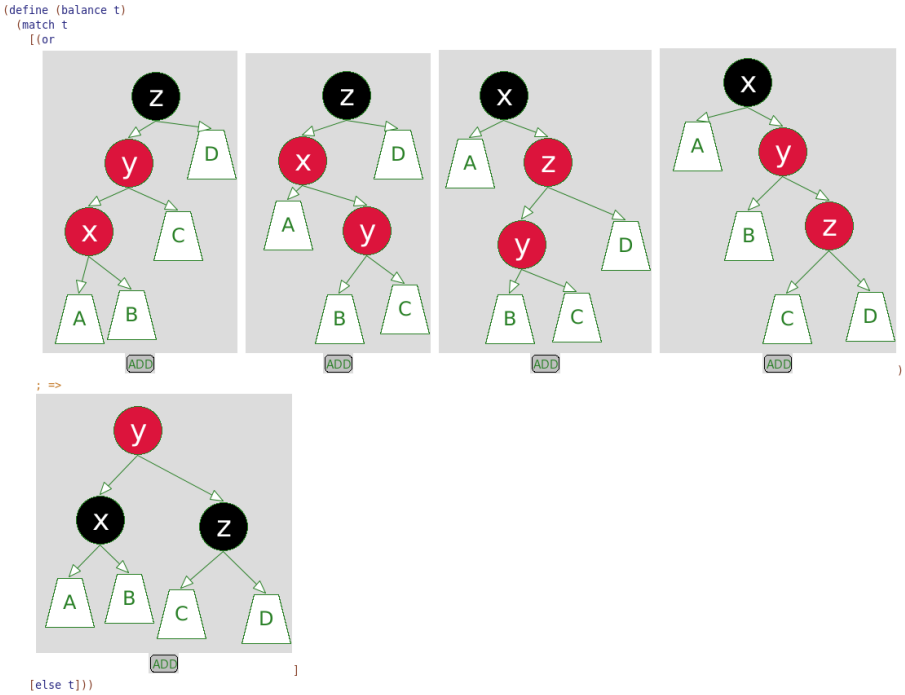

Figure 1: A definition of Okasaki's Red-Black Tree balancing algorithm that uses Andersen's Visual Interactive Syntax running in dr Racket

that one user, Cameron Moy, managed to use it to build visual syntax for PLT Redex programming language build in the Racket system.

3.3 Hazel

Hazel is a browser-based programming environment and a functional programming language developed to explore the notion of *typed holes*, where even incomplete programs can be properly typed, and thus various software services such as autocomplete can be implemented in a principled manner (Omar, Voysey, et al. 2016). Unlike Andersen, who focuses on adding visual extensions to textual code, the makers of Hazel decided to create a structure editor in such a way that the experience of using it resembles textual programming – but at every moment of program development the program code is properly structured (even if it contains holes).

Being of functional programming descent, Hazel is implemented in ReasonML, which is a functional language that can be compiled to native code and to JavaScript. The code expressed in Hazel is internally serialized to s-expressions, although this representation is never exposed to users, and could easily be changed to any other format.

Hazel comes with the concept of *live literals* or *livelits*, which are user-defined visual extensions (Omar, Moon, et al. 2021).

A live literal can be defined for some specific (concrete) type using the `livelit` keyword, and it consists of two type definitions (`Model` and `Action`) and four function definitions (`init`, `update`, `view` and `expand`).

Livelit definition is therefore very similar to the `define-interactive-syntax` form, where the `update` function corresponds to the `on-event` method, the `view` function corresponds to the `draw` method, and the `expand` function corresponds to the elaborator.

The creators of Hazel claim that the live literals are *compositional*, in the sense that it is possible to embed livelits inside other livelits. The compositionality of livelits is achieved via a mechanism called *splices*.

The paper which describes livelits presents the following examples: a slider, a color picker, a data frame (which is like a spreadsheet embedded in the source code), a specialized widget for adjusting school grade cutoffs (see figure 2), and another specialized widget for adjusting image filters (brightess and contrast).

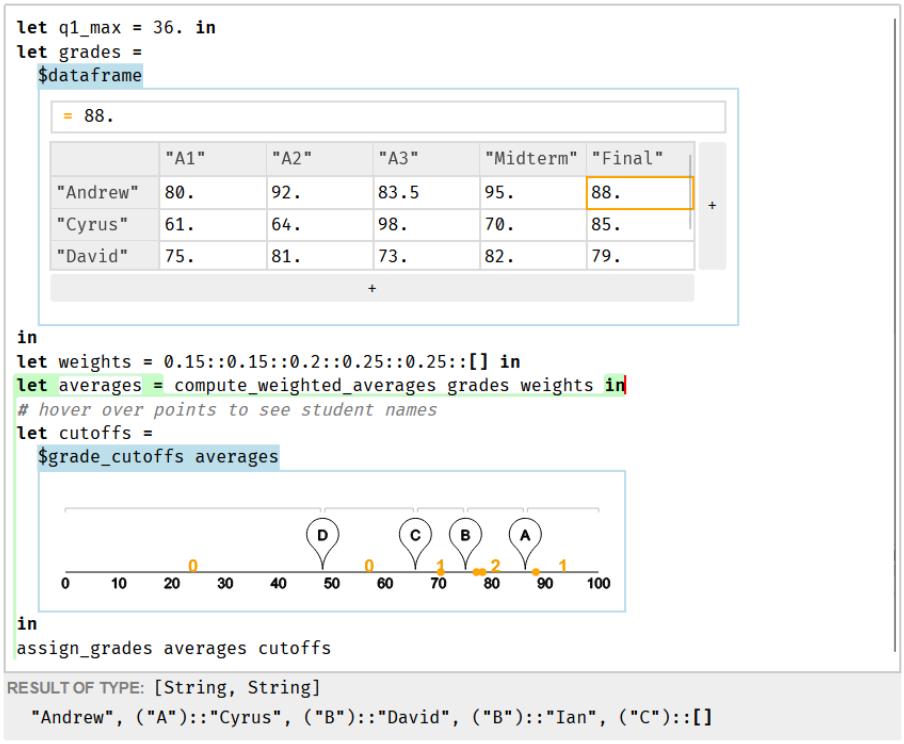

Figure 2: Examples of *livelits* in the Hazel programming system

However, the examination of the code base shows that most of those livelits are built-in (implemented in ReasonML), and therefore do not use the extension

system described in the paper. The only example of a livelit defined and used from Hazel itself available for examination is a variant of slider named `$slidy`. However, the exact extension method differs from the one described in the paper, and it resorts to techniques invoking inline JavaScript code.

Despite those apparent shortcommings, Hazel is a fine example of a General Visual Programming system. Although on the surface it may not seem as general as Interactive Visual Syntax, because it only allows to specify visual representations of literals (rather than arbitrary pieces of program), there is nothining in principle that should prevent programmers from e.g. defining literals that would express control flow diagrams, and then write interpreters for those diagrams (on the other hand, it isn't obvious whether it would be possible to use livelits as either patterns or templates in a pattern matching construct, as it was the case in Andersen's Red-Black Tree balancing example).

3.4 Polytope

Another General Programming System to consider is Polytope (Evans n.d.). Polytope is an editor written in TypeScript and running in the browser, which also provides fundamental architecture. The current implementation of Polytope is designed to work with JavaScript code. It comes with a bunch of specialized editors, including editors for directed and undirected graphs, math formulas, markdown and music staff. It should in principle be possible for developers to define their own editors and use them post hoc, but there is no distinct automatic mechanism for scanning and installing definitons in the system.

Editors are implemented by subclassing `EditorElement` (which itself is a subclass of browser's builtin `HTMLElement`), implementing `render` method (for displaying editor content on its HTML canvas or in another DOM element) and `toOutput` method (for converting an element to text, usually a JSON string) and registering event listeners (using the JavaScript's `addEventListener` method) and adding a parser/recognizer to the global `builders` array. Moreover, to make the extension available via a drop-down context menu, one needs to add an entry to the global `dropdownItems` array.

There is no universal mechanism that would connect the names of extensions with their occurrences within the source code. Instead, the particular extensions tend to check whether a considered object has some particular structure. For example, if a JSON object contains the keys `nodes` and `edges` (and their values are arrays), it will be considered a graph, and if it's an array and its first element is an object containing the key `note`, it will be considered music staff.

Polytope is also capable of typesetting certain mathematical functions, for example `exp(x, y)` can be rendered as x^y, and `div(x, y)` – as $\frac{x}{y}$. However, this feature doesn't appear automatically when the code is typed in the textual editor. It appears either when the code is loaded from a file, or when the user invokes the math editor.

The decision to base Polytope directly on the browser is a very interesting one, because it allows to reuse a lot of the existing infrastructure. On the other hand, it narrows the design space and constrains implementation techniques.

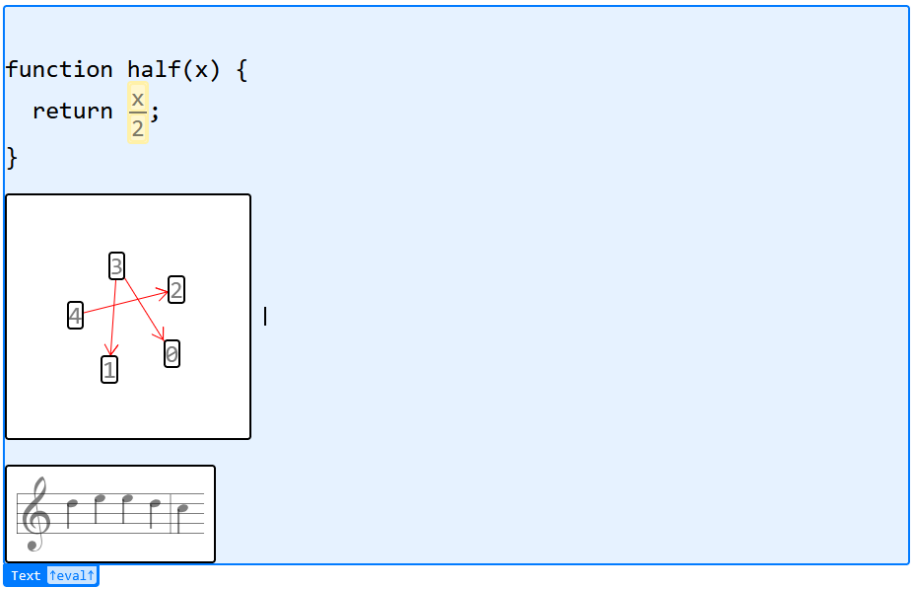

Figure 3: Some specialized editors that are available in the Polytope editor.

The editors in Polytope resemble Hazel's livelits in many ways, but the assumption to base the editing environment around text, rather than structure editing, is very much like in the Interactive Visual Syntax project.

3.5 JetBrains MPS

JetBrains MPS (MetaProgramming System) is a *language workbench* for creating domain-specific programming environments developed by the same company that develops and maintains numerous Integrated Development Environments for various programming systems, including IntelliJ IDEA, Android Studio, CLion, PyCharm, PhpStorm, WebStorm and a few others. The company also created the Kotlin programming language. Despite being developed by a commercial software company, MPS is available with full source code under the Apache License. It runs on the JVM platform.

MPS is a complex piece of software (JetBrains MPS Team 2025). One of its important components is a highly tweakable *projectional editor* which is able to operate directly on the edited program's abstract syntax tree, even though the process is made to resemble working with text. For this purpose, the editor has to understand the structure of a programming language that it's supposed to edit.

MPS uses a non-standard terminology to refer to the units from which a program is built (and it might therefore be misleading). In particular, the thing referred to as *a concept* in MPS roughly corresponds to *a grammar production rule* in the theory of parsing. However, MPS operates directly on the programs'

abstract syntax trees (which are called *models* in MPS terminology), so it is not a parser-based system – and in this sense MPS's concept can be thought of as *a syntax tree node type*.

The `.mps` files that are stored on the disk represent syntax trees that are serialized to a variant of the `XML` format, although they are neither readable to nor editable by humans (which probably defies the purpose of using XML for storing data).

The definition of a language in MPS consists of a few *aspects*. The most fundamental type is the language's *structure*, which consists of a set of language units (the aforementioned *concepts* or *syntax tree node types*). In other words, the *structure* aspect of a language definition determines which components are available in the language, and how they can be composed.

Another – closely related – aspect in MPS is called *constraints*, and this one – in turn – limits the way certain components can be composed.

A crucial – and very interesting – aspect of a language defined in MPS is called *editor*. It specifies the way the particular types of program nodes ought to be rendered. A seemingly fundamental language decision – whether to use a Python-style indentation-based syntax, or a Java-style syntax based on curly braces – in MPS is only a matter of tweaking an editor.

Editors in MPS don't necessarily have to be textual – according to the documentation, it is also possible to define editors that render arbitary graphics (based on the Java Swing API). This feature makes MPS a fine example of a General Visual Programming system.

That being said, the current set of practices focuses mainly on textual editors or some of their blends (such as decision tables and other tabular data, or matrix literals). A notable exception includes state machine definitions in the *mbeddr* project (Voelter et al. 2017).

Two aspects that are related to editing the language in the structure editor are called *actions* (for elaborating the syntax tree) and *intents* (for transforming sub-trees of the syntax tree).

Another aspect of language definition is *generator*, which determines how certain nodes are transformed to the constructs of some simpler language. In the LISP tradition, this step is usually called *macro expansion*. This aspect concerns transforming trees in one language into trees in another language. But in order to be able to integrate with some existing ecosystems, MPS also provides the notion of *base languages*, which can be transformed to text (the most typical base language of MPS `jetbrains.mps.baseLanguage`, which is in 1-1 correspondence to Java, and from which Java text can be generated).

The remaining aspects of language definition in MPS have to do with interfacing with its dataflow analysis engine and typechecking system.

At the moment of writing this, MPS has been available as a corporate-backed open source project for over 15 years. Yet despite a few documented use cases, its success seems very limited. This is probably due to the fact that the system itself seems to have a very steep learning curve, and it doesn't integrate well with existing tools (e.g. it is possible to use the `diff` tool on `.mps` files, but the information one can get from reading such diffs is close to none).

3.6 Glamorous Toolkit

Glamorous Toolkit is a *moldable* development environment developed by Tudor Girba and his associates (Nierstrasz and Gîrba 2024). It is based on Pharo Smalltalk and it inherits many properties that are typically associated with Smalltalk-based systems. In particular, it comes with a bunch of tools that can be used to explore the running system, and the system itself is organized as a multitude of communicating objects.

The key point of Glamorous Toolkit is the idea of *moldable development*, which is somewhat similar to language-oriented programming, but instead of designing domain-specific languages, it encourages developers to develop domain-specific (micro) tools for particular problems.

In the context of Glamorous Toolkit, this practice is often implemented by means of creating specialized *views* for particular objects.

Glamorous Toolkit is an open-source software, and its development model has been based on a private software consultancy that specializes in analyzing and improving existing legacy systems. Its developers also seem to put considerable effort into making the tool accessible to wider audiences. Yet it seems that despite all the effort, the learning curve remains fairly steep.

Glamorous Toolkit provides very good support for custom (user-defined) graphics rendering, and it comes with a very large number of examples.

However, in its current form, it can hardly be considered a General Visual Programming environment. The language used for program development – Smalltalk – is inherently textual, and there seems to be no attempts among the developers of Glamorous Toolkit to experiment with program representation; the main purpose of the custom visualizations is system analysis. This is not to say that Glamorous Toolkit wouldn't be the right substrate to develop a General Visual Programming system. Conversely, projects such as eToys (Kay et al. 1997; Kay 2007) or Sandblocks (Beckmann et al. 2020) have proven that Smalltalk-based systems offer a lot of flexibility when it comes to non-standard program representations.

4 Some Desirable Features of a Programming System

In the previous section, we have presented a number of existing systems, and we have analyzed them through the lens of General Visual Programming. The systems ranged from research prototypes to industrial-grade programming environments, but the systems which were more focused on visual programming tended to be more immature and less useful than the ones that treated programming more textually.

This is not to say that general visual programming is inherently worse or less capable than textual programming. But there is no doubt that purely textual programming has an uncomparably more established tradition. Moreover, text is both a very simple and a very elastic medium, and generic tools for working

with text (including text editors, but also operating system facilities such as a system clipboard which allows to perform copy and paste operations between different applications) often turn out to be sufficient for developing computer programs.

Lastly, the paradigm which treats programs as *plain text* is agnostic of any particular authoring tools, so it decouples the act of creating a program from the act of interpreting it, effectively allowing programmers to use the editors of their preference. It also allows to develop programming language tools independently of editors.

In this sense, textual programming is on a privileged position compared to visual programming. It seems that, if we are to develop a compelling visual programming system (that could compete with the existing textual programming systems), we somehow need to tackle the question of universality: is it possible to construct a generic hybrid visual-textual editor that would support a wide variety of programming languages, the same way a generic text editor supports them?

The alternative is to focus on a single specific programming language (possibly even to design a new one), which may simplify the work greatly, but it can also limit the potential audience of the system. It may also not be immediately clear how to choose the particular programming language – by the merits of popularity, expected demand for visual extensions, personal preference, simplicity of implementation or some other reason.

Of course, the intrinsic qualities of a software tool are not enough to determine its success – even the highest-quality seed needs the right soil in which to take root and flourish.

The single most important feature a system must possess in order to become successful is that it addresses the actual needs of its potential users. It must also be simple enough to be understandable, or at least it must provide some learning path that eventually leads to sufficient understanding.

5 The GRASP System

This section describes our experience with the development of the GRASP programming system.

Like most systems described in this paper, GRASP is a General Visual Programming system. It is neither a text-based system (such as Polytope or Visual Interactive Syntax) nor a projectional editor (such as MPS or Hazel). Instead, it is a generic structure editor for s-expressions, so while it allows to create malformed programs (or rather structures), the textual representation of those structures necessarily consists of balanced parentheses.

GRASP renders a balanced pair of parentheses as a *box*.

For example, a definition of the *factorial* function that uses the GRASP representation is depicted on figure 4.

Boxes in GRASP occupy a rectangular area. The left parentheses can be used to drag a given box out of the document and move it elsewhere, and the

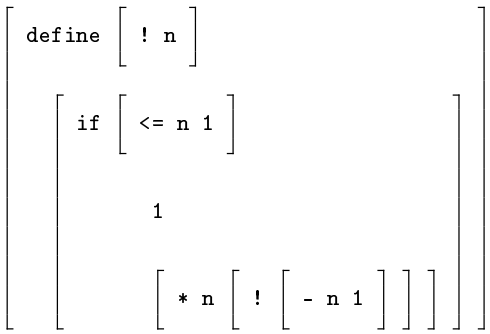

Figure 4: A definition of the factorial function rendered using the GRASP representation

right parentheses are used to resize the box.

GRASP also allows users to define *extensions* or *magic boxes*, which are an instances of the domain-specific editors discussed in the earlier section.

5.1 A Brief History

At the time of writing, GRASP has gone through four iterations of development. The first prototype was written in Racket, and its development was preceded by announcing the *Draggable Rectangle Challenge* on the Quora social network[2], which consisted of three tasks:

1. Implement a program where I can freely drag rectangles around the screen
2. Implement a program like above, but the rectangles can also contain draggable rectangles (and so on)
3. Implement a program where I can move the rectangles from one "inner rectangle" into some others.

The purpose of the challenge was to explore alternative designs for the fundamental mechanics of GRASP editing.

The key tension was between functional programming (immutable) designs and object-oriented (mutable) ones.

5.2 The first prototype (2018-2019)

The initial implementation in the Racket programming language[3] explored the technique of *mix-in composition*, where all the features were implemented as

[2] `https://eidolon-language.quora.com/Draggable-rectangle-challenge-part-I-the-introduction`

[3] the source code of the prototype and the solutions to the Draggable Rectangle Challenge is available at `https://github.com/panicz/sracket`

mix-ins. This technique, however, turned out to be problematic for several reasons. First, because we relied on the identity of closures of the Scheme programming language as the identity of objects, and each mix-in created a new closure, the identity of objects was inconsistent. This could probably be improved by modeling object identity more explicitly. The other problem was that each mix-in created a new layer of indirection in runtime, which – without a proper language support – incurred a performance penalty.

While the prototype allowed to perform drag&drop and resize the boxes, its editing capabilities were very limited. It was not possible to open or save files, and while the elements could be removed, the only kind of elements that could be added were boxes (using the `[` key).

The content of the screen could not be scrolled, and the support for keyboard editing was very limited. Nevertheless, even though both the content of the edited document and the list of supported visual extensions were hard-coded in the editor, the extension mechanism was already designed to be convenient for users: it simply required to create an object that is able to understand the messages `as-image` (which is supposed to produce a Racket image) and `as-expression` (which is supposed to return a list that stands for an expression).

Registering an extension is based on a special form called `define-interaction`.

The entire prototype consisted of less than 1,000 lines of code, and it came with a built-in extension for rendering directed graphs (see figure 5 for its definition).

5.3 The second prototype (2020)

The choice for the Racket programming language certainly had its merits. Racket fits naturally with an editor that operates on s-expressions, and it is maintained for the three major platforms (Windows, Linux and OS X).

However, Racket also has its limitations. In particular, it does not support the creation of applications for the Android platform, which was of our interest, because despite being very popular, it offered very few programming tools, most of which were scaled-down versions of desktop applications, and none of which tried to explore the specifics of programming that would involve small multi-touch screens.

For this reason, we decided to develop the next prototype in Java, which has been the recommended language for Android development since the inception of the platform.

After doing some research, we made the decision to not only develop the next version of GRASP for Android, but also – to develop it **on** an Android phone, using the Termux environment. There were several reasons for this decision. This mode of development provided a faster feedback and better continuity than having to attach a phone to a desktop computer after each modification of the application. But more importantly, the constraints of the environment were forcing us to think about the editor features that would ease the development in the future.

```
(define (Graph neighbour-list)
  (let* ((vertices (in-circle
                    (map (lambda (`(,node . ,neigbours))
                           (Vertex node))
                         neighbour-list)))
         (collection (Collection vertices)))
    (lambda message
      (match message

        (`(as-image)
         (let ((image (collection 'as-image)))
           (for `(,source (,node . ,neighbours)) in (zip vertices
                                                         neighbour-list)
             (for neighbour in neighbours
               (let ((target (find (lambda (v)
                                     (eq? neighbour (v 'as-expression)))
                                   vertices)))
                 (draw-edge! source target image))))
           image))

        (`(as-expression)
         `(digraph . ,neighbour-list))

        (`(acquire-element!)
         #false)

        (_
         (apply collection message))))))

(define-interaction (digraph . neighbour-list)
  (Graph neighbour-list))
```

Figure 5: A code that defines the `digraph` extension in the first prototype of GRASP

The second prototype was rather clumsy. It used a box representation that was similar to the one employed by the Racket prototype. It did support drag&drop and the creation of boxes, as well as some rudimentary text editing, but it wasn't able to persist the created structures, nor to evaluate them. Despite having some placeholders in the user interface, it did not support any extensions.

The main purpose was to check whether editing code on a multi-touch screen was feasible. The most valuable artefact from the development is a three-minute video showing the construction of a definition of the factorial function[4]. The fact that it took entire 3 minutes was rather alarming (entering the same text using a traditional keyboard takes about 15 seconds, or about 30 seconds using an

[4] `https://www.youtube.com/watch?v=BmZ39IfElzg`

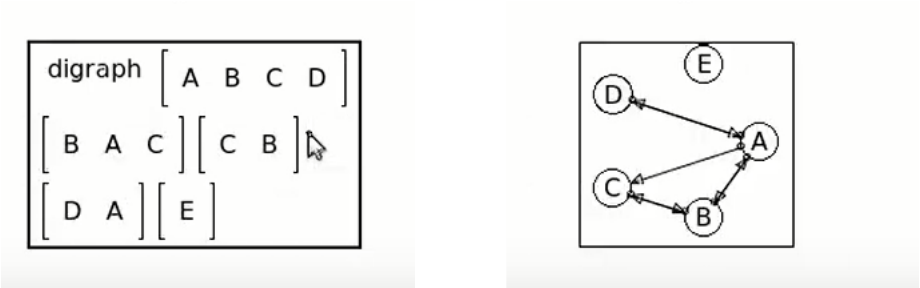

Figure 6: A structural representation of a directed graph and its visualization in the first prototype of GRASP

on-screen keyboard on a phone), but it also encouraged to look for optimizations in the editing process.

Despite being in some ways less capable than the first prototype, the second prototype consisted of almost 3,000 lines of Java code.

5.4 The third prototype (2020-2021)

Architecture-wise, the second prototype turned out to be a dead-end.

The third prototype was planned in stages, to allow to backtrack on bad design decisions without having to start everything over.

The initial stages for the third prototype featured splitting the screen into multiple panes, as is possible in such editors as Emacs, vi or Blender 3D. The documents visible inside the panes could be scrolled and rotated (using pinch-zoom).

It was also the first prototype of GRASP which supported loading and saving s-expression text files (but the supported syntax only included lists and atoms – no strings, vectors or comments). The editing capabilities were also vastly improved: the left parenthesis was now responsible for dragging expressions, and the rigth one – for resizing them. The expressions could be removed by throwing them quickly off the screen, and the attempt to throw the right parenthesis off the screen would cause the content of the list to be spliced into its parent.

Double-tapping on the left parenthesis would make a copy of the entire expression.

The third prototype of GRASP also supported several gestures: drawing a rectangle would add a new box to the document, drawing a spike would cause an expression under the spike to be evaluated, and drawing an underscore gesture (down-right-up) would add a new atom. Adding a new atom involved opening a floating text editor window. This prototype was also capable of reading the data from the phone's inertial measurement unit, and use the act of shaking to invoke code reindentation.

The support for those gestures allowed to reduce the time required to define the `factorial` function from 3 minutes to about 1 minute after some training,

and (according to the only known user, who also happens to be the developer of GRASP) the editor itself was fairly pleasant to use, even despite lack of support for operations such as *undo* or *copy-paste* (Godek 2023).

The document representation could be described by the following Haskell-like type definitions:

```
Bit = Atom following_space: (Space|null)
           text: String
    | Box following_space: (Space|null)
          first_interline: Interline;

Space = Space width: float
              following_bit: (Bit|null);

Interline = Interline height: float
                      following_line: (Line|null);

Line = Line first_space: (Space|null)
            next_interline: (Interline|null);
```

so `Bit` was an interface with two implementations – `Box` and `Atom`, where a `Box` consisted of `Interline`s interleaved with `Line`s, and each line consisted of `Bit`s interleaved with `Space`s.

Because of its complexity, this representation wasn't particularly pleasant to work with, and it was hard to integrate it with any extension system (although with some effort we managed to create an extension which allowed to present a green rectangle).

Despite having been planned in stages, the third prototype had some flaws that were hard to backtrack. After integrating with the Kawa evaluator, the build times became much longer and were often using up all the available memory. Moreover, the third prototype only ran on Android, and testing changes required the full build-install cycle.

But the bigger issue was that the Java programming language, which was used for the development, wasn't based on s-expressions, which meant that there would be no option to use GRASP for the further development of GRASP. This reason alone was a sufficient to drop almost 10,000 lines of Java code and start over.

5.5 The fourth prototype (2022-2025)

The fourth prototype was started after a successful attempt to use Kawa Scheme to create an application for Android. This time the editor was intended to run (and be developed) not only on Android, but also on the PC. An additional requirement was to be able to run various pieces of code without rebuilding the entire project. Since GRASP is a visual and interactive application and the support for graphics in terminal-based environments is rather poor, we decided that we also want to be able to render GRASP's "graphical" output to text, and also to run the entire GRASP application in a terminal emulator.

This decision had significant implications on the architecture of GRASP. All the rendering in the system is performed through an interface called `Painter`. The code base of GRASP provides four implementations of that interface: one for Android, one for desktop, one for terminal, and one for string rendering (the latter two share a lot of their implementations).

Excluding tests, there are three main entry points to the fourth prototype of GRASP – one for the desktop client, one for the terminal client, and one for the Android client. Each of those clients also uses different code for handling events, but from the perspective of GRASP itself, these are unified.

The decision to make GRASP available on PCs was not driven solely by the desire to make testing more efficient. Just as an Android device is a natural environment for developing applications intended to run on Android and make use of multi-touch interaction, a PC is the natural platform for building applications controlled primarily via keyboard and mouse. One of the design goals for the fourth prototype of GRASP was to make keyboard editing at least as efficient as in a traditional text editor – and we believe that this goal has largely been met.

A few words need to be said about the Kawa language. Before developing the fourth prototype of GRASP, we had over ten years of experience with programming in Scheme (mainly Guile and Racket), and while we were mostly enthusiastic about the language, we felt that it lacked some important features, such as type signatures or robust record type definitions.

To our surprise, the Kawa language either resolved these deficiencies (by providing optional type annotations for variables and procedure return values) or offered mechanisms that allowed us to resolve them ourselves. It also came with very good JVM interoperability and a mostly tasteful (in our opinion) set of syntactic extensions.

On the other hand, during development we encountered several issues stemming from the behaviour of the Kawa compiler. The available community support was limited, mainly because Kawa is no longer actively maintained, which made it difficult to resolve problems or obtain clarifications.

Overall, despite these problems, we feel that the combination of Java-like, strongly typed, interface-based object orientation with the syntactic flexibility and minimalism of the Scheme programming language was an excellent match for building the GRASP project.

The fourth prototype of GRASP consists of over 30,000 lines of code, making it a fairly large project. It is also the most feature-complete version to date – it supports most of the syntax of the Scheme programming language, including not only lists and atoms, but also strings and three kinds of comments defined by the Scheme language specification (line comments, block comments and expression comments).

It also includes a working mechanism for writing extensions (described in more detail in the next section), and at the time of writing the repository contains over a dozen different extensions, including buttons, a visual stepper (Godek 2024) and a simple two-dimensional physics engine (which is capable of using the phone's accelerometer for accessing gravity information).

The representation of documents in the fourth prototype of GRASP is somewhat convoluted: initially we tried to design an editor that would operate directly on Lisp's `cons`-cells. This was a bit challenging, because the document must also store information about whitespaces and comments. We therefore decided to use a few hash tables, called `pre-head-space`, `post-head-space`, `pre-tail-space` and `post-tail-space` to store the whitespace/comment information. There was an additional table called `dotted?`, which contained `#true` if a given cell was written down using the dotted-pair notation. Moreover, since there is only one instance of an empty list in most Lisps (including Kawa), but lists can also contain whitespace and comments, we created additional pair of tables – `null-head-space` and `null-tail-space` – to store the whitespace/comment information if, respectively, a head or a tail of the cell contained a reference to the empty list.

The use of hash tables turned out to be problematic, because the developers of Kawa decided to override their `cons`-cell's `equals` method with Scheme's `equal?`-like semantics, which caused all lists with identical contents to be treated as the same list instance.

If we wanted to stick with our design, we therefore either had to patch the Kawa compiler, or to subclass its class that represents `cons`-cells. After running some experiments, we chose the latter option.

The complication grew even further when we started implementing atom editing capability. There were a few problems that we were facing:

- since most atoms (e.g. symbols or numbers) are immutable, every keystroke would have to generate a new object
- during editing, certain kinds of atoms would need to be transformed into other kinds of atoms (e.g. numbers into symbols)
- certain atoms can have many different representations (for example, numbers can be represented as decimal, hexadecimal, octal or binary)

For this reason, we decided to create a mutable proxy called `Atom`, that could stand for different kinds of atoms (symbols, numbers, booleans). This change had an unfortunate impact, though: the parts of the document could no longer be interpreted by the Kawa evaluator.

To account for that, we introduced a *parameter object* called `cell-access-mode`, which could take one of the two enum values (`Editing` or `Evaluating`) and we further modified our subclass of Kawa's `cons`-cell class: its `getCar` and `getCdr` methods checked the value of the parameter, and either returned the `Atom`, or the result of invoking `read` on the `Atom`'s content.

In addition to lists and atoms, GRASP provides a distinct representation of string objects. Strings are deliberately rendered in a monospace font, enabling textual representation of GRASP objects to be displayed consistently in graphical clients.

Another kind of primitive objects that can be present in the document are spaces. A `Space` object holds a list of integer numbers that represent single

spaces. Two consecutive integers in the list represent a line break. The list can also contain `Comment` objects, which can be of three types: `LineComment`, `BlockComment` and `ExpressionComment`. A `LineComment` also induces a line break.

The fourth prototype also features clipboard integration[5] and the *undo* mechanism.

Undo is implemented by tracking invertible operations on the mutable document object, and it conceptually maintains a history tree (although currently it isn't visualized in any way, and the part of the undone history becomes inaccessible after new operations).

The implementation of keyboard editing requires tracking cursor position. This task is fairly easy for regular text editors, where a cursor can be represented by a line number and a column number, but it becomes more complicated for a structure editor.

In the case of GRASP, a cursor is represented by a sequence of indices, where the length of the sequence depends on the nesting level of the expression. One challenging aspect in the case of GRASP is the vertical cursor movement, for which no satisfactory solution has yet been found.

Like the third prototype, the fourth prototype implements screen splitting by means of gestures. Other kinds of gestures are currently not supported.

We made several attempts to use the fourth prototype of GRASP to further develop GRASP. In theory that should be possible, but in practice we found several shortcomings that were severely impeding the process. The biggest problem was that the editor was fairly slow when handling large files. One feature that we considered essential for code editing was the *search* facility – which we implemented, but we feel that the textual search is somewhat limiting for a structure editor. Moreover, because the representation of the document was so convoluted, some editing operations were not implemented correctly, which made the use of the editor somewhat frustrating. Nevertheless, we managed to use GRASP on the phone successfully to develop solutions to some of the 2024 Advent of Code riddles.

6 The Architecture of GRASP

This section is based primarily on our work on the fourth prototype of GRASP, but it describes architectural decisions that we expect to be preserved in all future versions of the system.

The most fundamental component of GRASP is the `Element`. `Element`s come in two basic categories: `Space`s and `Tile`s. The main difference between them is that `Tile`s are rectangular objects (and therefore have a width and height), whereas `Space`s are only used to ensure *textual ordering* between elements.

[5] For some reason, the terminal client is unable to connect to the system clipboard, but it maintains its internal clipboard.

All `Elements` support cursor navigation (moving the cursor index forward or backward) and allow retrieving the element located at a given cursor index. They can also be rendered as part of the document.

The document model in GRASP differs significantly from the one that is widely known from the browser environments in that its elements do not store a pointer to their parent, and they are always positioned with respect to their parent (so they cannot have *absolute* position).

GRASP provides several primitive implementations of the `Tile` interface — most notably those representing boxes, atoms, strings and block comments.

Users can also define their own implementations of the `Tile` interface through the *extension mechanism*. User-defined extensions provide their own rendering procedures, and they are typically `Interactive`, meaning that they can respond to user input (keystrokes and mouse/pointer events). The combination of the `Interactive` and `Tile` interfaces is called `Enchanted`.

There are two fundamental ways to define extensions. The first requires implementing all necessary methods (for drawing and input handling). The second allows constructing extensions from simpler extensions using *combinators*.

Figure 7 shows the definition of `PlayerWithControls`, which uses three combinators: `bordered`, `beside`, and `below`.

```
(define-interface Player (Enchanted Playable Animation))

(define (PlayerWithControls player::Player)::Enchanted
  (bordered
   (below
    player
    (beside
     (Button label: "|◁◁"
             action: (lambda () (player:rewind!)))
     (Button label: "|◁ "
             action: (lambda () (player:back!)))
     (Button label: (lambda ()
                      (if (player:playing?) "| |" " ▷ "))
             action: (lambda ()
                       (if (player:playing?)
                           (player:pause!)
                           (player:play!))))
     (Button label: " ▷|"
             action: (lambda () (player:next!)))
     (Button label: "▷▷|"
             action: (lambda () (player:fast-forward!)))))))
```

Figure 7: An example of combinator usage (`bordered`, `beside`, `below`) in the definition of another combinator

This definition is in turn used to implement the `Stepper` widget, an instance of which is shown in Figure 8.

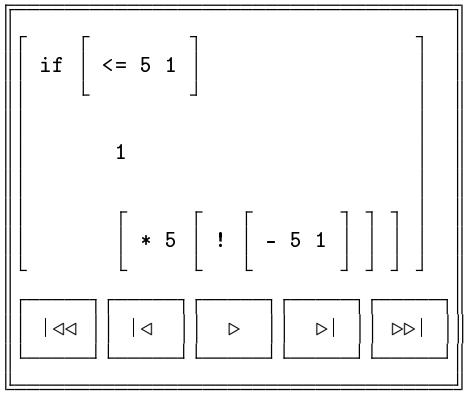

Figure 8: `PlayerWithControls` used in the implementation of the visual `Stepper`

Most visual components which are used in the interface of GRASP itself (such as file lists, buttons, text fields) are also GRASP extensions, and as such they can be embedded in the GRASP document.

Currently every extension must have a name, although this might change in the future. In addition to being a `Tile`, an extension should know how to serialize itself to an s-expression. However, when the editor turns an s-expression into a *magic box*, it remembers the original s-expression, so that certain extensions which consist of non-serializable components (such as closures) can be properly serialized.

Unfortunately at this point there is no mechanism that would force all extensions to behave properly with regard to their serialization, and some extensions are created by explicitly calling `eval`, which is a well-known potential security vulnerability.

During the operation of GRASP, there is a Scheme interpreter running in the background. The first thing that the interpreter does is interpret the expressions from the `init.scm` file[6].

The `init.scm` file is responsible for setting up the key bindings, and for setting the content of the `screen`.

By default, the `screen` contains an `Editor` or – if the user decides to split the `screen` – a `Split`. However, it is possible to set any user-defined extension that is `Maximizable`. The idea behind this mechanism is to turn GRASP into an *application platform*, rather than merely an editor[7].

We mentioned in the previous section that rendering in GRASP is mediated through the `Painter` interface. The `Painter` interface is designed to account for

[6] The current implementation of GRASP is typically bundled in a `.jar` or `.apk` file, which are actually `.zip` archives, and the `init.scm` file is stored in its `assets` subdirectory

[7] As a proof of concept, we developed a simple game for kids that uses Android *Text to Speech* and *Speech to Text* services, where the player assembles uttered words from letters.

the capabilities and limitations of character terminals, so for example it provides different routines for rendering lines that are perpendicular or parallel to the screen, and different ones for drawing lines that can be skewed. It also has very limited options regarding font rendering.

At the time of writing, the `Painter` interface consists of over 100 methods, and it needs to be simplified.

`Painter` is also capable of handling `Animation`s. `Animation` is an interface that has one method, `advance!`, which accepts a time step in milliseconds and returns a Boolean value – `#false` if the animation has finished, and `#true` if the animation should continue.

GRASP also features a `TouchEventProcessor` akin to Android's `GestureDetector` (for detecting events such as double tap or long press), but designed so that it can be used also by desktop and terminal clients.

7 Discussion and Future Work

We began by describing the concept of General Visual Programming and analyzing various existing GVP systems. However, we did not explain why such systems might be worth pursuing.

The fundamental reason behind our developments is simply to make the implementations of computer programs more understandable. This goal cannot be achieved by merely using even the finest tools – it also requires programmers to apply deliberate practices whose purpose is to enhance the clarity of the source code of their programs.

In the course of our work we observed that various parts of a computer program can differ drastically with regard to their ease of understanding, and that – in particular – it is possible to write code that clarifies some other code. Consider, for example, the following definition in the Scheme programming language:

```
(define (f lol)
  (apply map list lol))
```

We've found that even though the body of the definition consists only of four symbols, even programmers who are presented with the meanings of the functions `apply`, `map` and `list` find it troublesome to understand the intent behind that definition.

However, if we present them with a usage example such as

```
(e.g. (f '((a b c)
           (d e f))) ===> ((a d)
                           (b e)
                           (c f)))
```

the intent becomes much clearer to them (those of them who went through a course in linear algebra are typically also able to suggest the name `transpose`

instead of `f`, which shows that naming is also a technique of making programs clearer).

The difference between the definition of `f` and its example usage is that the first one is *abstract* and introduces a new concept into the system, while the second is *concrete* and it only puts certain content in front of the mind's eye.

Professional programmers have a habit of *writing tests* for their code. This name is unfortunate, and it shapes our expectations poorly: we test things in order to find out how they work (or *if* they work the way we think they should). Of course, there is tremendous value in having automated *integration tests* for complex systems, but many *unit tests* play mainly the role of *system documentation* or a *learning resource* for programmers to figure the system out without even running the code.

The `transpose` example may seem appealing, but that's mainly because its domain is naturally expressed in text. However, there are many domains – such as graphs or geometry – that do not share this property. A convex hull algorithm could be *unit-tested* using a set of pairs of numbers, but those numbers would likely not appeal to the imagination of the readers of the code the same way a picture of those points would.

Similarly, consider the following example of the *A* search algorithm* (herein called `optimal-path`):

```
(e.g.
 (let ((graph-from-Wikipedia '((start (a 1.5) (d 2))
                               (a (b 2) (start 1.5))
                               (b (c 3) (a 2))
                               (c (end 4) (b 3))
                               (d (e 3) (start 2))
                               (e (end 2) (d 3))
                               (end (c 4) (e 2))))
       (heuristics '((a . 4)
                     (b . 2)
                     (c . 4)
                     (d . 4.5)
                     (e . 2)
                     (end . 0))))

   (define ((callable alist) key)
     (assoc-ref alist key))

   (optimial-path on: (callable graph-from-Wikipedia)
          from: 'start
          until: (is _ equal? 'end)
          guided-by: (callable heuristics)))
 ===> (start d e end) 7)
```

This example is quite complex; even understanding what it is about requires effort. It would be much better if we could just see the graph and the path on the graph – or better yet, to see how the algorithm operates step by step.

Our hope is that we will be able to adjust our visual stepper to make it possible to look at algorithms that operate on the *natural representations* of data structures, and that we will be able to employ animations for observing the operations of algorithms, without paying much overhead in the development time.

But in order for this goal to be attainable, a general visual programming system should be simple in its construction and no more difficult to use than conventional text-based systems.

With this regard, there is still some work to be done. Probably the most important is the simplification of the document representation, which – being based on Lisp's linked lists – is quite convoluted. For the next iteration, we plan to represent expressions using a pair of arrays: the first one, containing n elements, would contain `Tile`s, and the second, containing $n + 1$ elements, would contain `Space`s. (We call this a `Comb` representation, because it makes the tiles resemble comb teeth, but also because it represents *combinations* of expressions.)

Another important aspect is the extension system. Although the current concept of extensions in GRASP is satisfactory, the way extensions are defined by users could be significantly improved – with regard to both ergonomics and safety.

Lastly, although the most recent prototype of GRASP is able to load and save Scheme files, there is a sort of *impedance mismatch* when it comes to the preferred *indentation style*. More specifically, unlike the editors that operate on text, GRASP prefers very little to no indentation, because the *nested boxes* representation already works like a sort of an indentation, and the code that uses the traditional Lisp-style indentation looks *too indented* in GRASP.

Obviously this could be solved by removing indentation when a file is loaded and adding it back when it is saved, but it would work only for files which use standard indentation. Otherwise using GRASP to load and save a file without any edits would mess up the indentation.

So far we've been mainly considering the *close-up* perspective on programming – which is the usual perspective when people discuss programming tools. But the pinch zoom that is present in GRASP also provides a fairly unusual way of looking at code – it allows to smoothly zoom out to see entire documents at once. Unfortunately, these zoomed-out views are not very informative: they typically appear as vertical white stripes with some illegible dark spots on them.

This is particularly frustrating when we think that it is possible to construct a *bird's eye view* of a program module that would actually be useful, and that such view could contain a class diagram or function signatures, devoid of their actual implementations.

And when we zoom out even further, we should no longer see a single module, but instead we should see the modules from the entire project.

Currently this is usually done by means of a file browser. But this representation usually shows a module as a generic icon with a file name, residing in a directory tree. While this representation gives some information, there is a lot of information that it misses. In particular, the size of the icon could

depend (logarithmically or otherwise) on the size of the module. Moreover, in module systems that xpreclude circular dependencies, the modules that make up a project naturally form a layered graph, where the topmost modules are the ones that have no dependencies, and the bottommost ones usually contain application-related code. It would also be desirable to be able to highlight all modules that a given module depends on, as well as all the modules that depend on a given module (but in order to remain legible, this visualization would need to be dynamic)[8].

Another thing that might be important for the adaption of the concepts presented here is the availability of systems which support them.

In particular, integrating GRASP with web browsers would make it possible to have GRASP snippets (including extensions such as its visual stepper) available on web pages.

There is also a number of smaller things that could be added to or improved in the system. For example, it could be handy to have a generic object inspector, which would allow to add arbitrary Java objects to the document. Another nice feature would be an integration of GRASP with a popular version control system such as git, which also has a great potential of interesting and valuabe visualizations[9].

But, in order for those developments to make sense, GRASP must first become sufficiently usable and robust to acquire a user base.

References

Andersen, Leif (2022). "Adding Visual and Interactive-Syntax to Textual Programs". PhD thesis. Northeastern University. URL: https://www2.ccs.neu.edu/racket/pubs/dissertation-andersen.pdf.

Andersen, Leif, Michael Ballantyne, and Matthias Felleisen (2020). "Adding Interactive Visual Syntax to Textual Code". In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA, pp. 1–28. DOI: 10.1145/3428290.

Andersen, Leif, Cameron Moy, et al. (2024). *Making Hybrid Languages: A Recipe*. arXiv: 2403.01335 [cs.PL]. URL: https://arxiv.org/abs/2403.01335.

Beckmann, Tom et al. (2020). "Visual design for a tree-oriented projectional editor". In: *Proceedings of the 4th International Conference on the Art, Science, and Engineering of Programming (Programming'20 Companion)*, pp. 113–119. DOI: 10.1145/3397537.3397560. URL: https://doi.org/10.1145/3397537.3397560.

[8] This idea is reminescent of the *C4 model* developed by Simon Brown (Brown 2019), as well as the demo from Ivan Daniluk's 2019 GopherCon Europe talk titled *Rethinking Visual Programming* (Daniluk n.d.).

[9] There is a number of code quality assessment techniques developed by Adam Tornhill that were inspired by forensic analysis (Tornhill 2015)

Brown, Simon (2019). *Software Architecture for Developers*. Contains the formal definition of the C4 model. Leanpub. URL: https://softwarearchitecturefordevelopers.com/.

Daniluk, Ivan (n.d.). *Rethinking Visual Programming with Go — talk at GopherCon Europe 2019*. YouTube video. URL: https://www.youtube.com/watch?v=Ps3mBPcjySE, Accessed: 2025-12-08.

Evans, Elliott (n.d.). *Polytope: Experimental Recursively-Embeddable Code Editor*. https://elliot.website/editor/. Accessed: December 5, 2025.

Felleisen, Matthias, Robert Bruce Findler, et al. (2018). "A Programmable Programming Language". In: *Communications of the ACM* 61.3. DOI: 10.1145/3127323.

Felleisen, Matthias and PLT Team (2025). *2htdp/Universe Teachpack in Racket*. Accessed: December 5, 2025. URL: https://docs.racket-lang.org/teachpack/2htdpuniverse.html.

Findler, Robert Bruce and PLT (2010). *DrRacket: Programming Environment*. Tech. rep. PLT-TR-2010-2. URL: https://racket-lang.org/tr2/. PLT Design Inc.

Flatt, Matthew, Robert Bruce Findler, and Matthias Felleisen (2006). "Scheme with Classes, Mixins, and Traits". In: *Lecture Notes in Computer Science*. Springer Nature, pp. 270–289. DOI: 10.1007/11924661_17.

Fluxus: Livecoding and Rapid-Prototyping Environment (n.d.). https://www.pawfal.org/Software/fluxus_/. Accessed: December 5, 2025.

Foundation, Scratch (2024). *Scratch Annual Report 2024*. https://annualreport.scratchfoundation.org/. Accessed: 2025-12-08.

Godek, Maciej (2023). "GRASP: An Extensible Tactile Interface for Editing S-expressions". In: *Proceedings of the 16th European Lisp Symposium (ELS'23)*. DOI: 10.5281/zenodo.7816633.

— (2024). *An Implementation of a Visual Stepper in the GRASP Programming System*. arXiv: 2508.04859 [cs.PL]. URL: https://www.arxiv.org/abs/2508.04859.

JetBrains MPS Team (2025). *MPS User's Guide*. Accessed: December 8, 2025. URL: https://www.jetbrains.com/help/mps/mps-user-s-guide.html.

Kay, Alan (2007). *Children Learning by Doing: Etoys on the OLPC XO*. Tech. rep. RN-2007-006-a. Accessed: December 8, 2025. Viewpoints Research Institute. URL: https://worrydream.com/refs/Kay_2007_-_Children_Learning_by_Doing.pdf.

Kay, Alan et al. (1997). "Etoys and Sim & Stories". In: *VPRI Paper for Historical Context*. Accessed: December 8, 2025. URL: https://tinlizzie.org/VPRIPapers/hc_etoys_sim_1997.pdf.

Kernighan, Brian W. (2019). *UNIX: A History and a Memoir*. Independently published.

LabVIEW Documentation (2025). Accessed: 2025-12-08. National Instruments. URL: https://www.ni.com/labview.

Nierstrasz, Oscar and Tudor Gîrba (2024). *Moldable Development Patterns*. arXiv: 2409.18811 [cs.SE]. URL: https://arxiv.org/abs/2409.18811.

Omar, Cyrus, David Moon, et al. (2021). "Filling Typed Holes with Live GUIs". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21)*. Virtual, Canada: ACM, pp. 511–525. DOI: 10.1145/3453483.3454059.

Omar, Cyrus, Ian Voysey, et al. (2016). "Hazelnut: A Bidirectionally Typed Structure Editor Calculus". In: *arXiv preprint arXiv:1607.04180*. Last revised: February 4, 2019; Accessed: December 5, 2025. URL: https://arxiv.org/pdf/1607.04180.

Resnick, Mitchel et al. (2009). "Scratch: Programming for All". In: *Communications of the ACM*. Vol. 52. 11, pp. 60–67. DOI: 10.1145/1592761.1592779.

Simulink Documentation (2025). Accessed: 2025-12-08. MathWorks. URL: https://www.mathworks.com/help/simulink.

TIOBE Index (2025). Accessed: 2025-12-08. TIOBE Software. URL: https://www.tiobe.com/tiobe-index/.

Tornhill, Adam (2015). *Your Code as a Crime Scene: Use Forensic Techniques to Arrest Defects, Bottlenecks, and Bad Design in Your Programs*. Pragmatic Bookshelf. ISBN: 9781941222294.

Unreal Engine: Blueprint Visual Scripting (2025). Accessed: 2025-12-08. Epic Games. URL: https://docs.unrealengine.com/Blueprints.

Voelter, Markus et al. (2017). "Lessons Learned from Developing mbeddr: A Case Study in Language Engineering with MPS". In: *ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS'17)*, pp. 585–630. URL: https://link.springer.com/article/10.1007/s10270-016-0575-4.