The application of functional programming to the construction of a visual programming environment

Łukasz Czapliński

September 5, 2016

Abstract

Traditional programming methods are becoming increasingly disconnected from modern input methods, like touchscreens. An alternative — programming with diagrams, called Visual Programming Languages, has long existed, but received little attention. This thesis uses ideas inspired by functional programming to propose an approach to representation of Clojure programming language, that allows manipulation of program both as a text and as a diagram. This approach is implemented as Visual Programming Editor — Jarvis.

1 Introduction

1.1 Visual Programming Languages (VPLs)

Programming is an art of communication — both with a machine and with other people. To express algorithms in a machine-readable way, we use programming languages. When it comes to describing our ideas to other programmers, we often use entirely different means. Diagrams are one of the most universal ones. Since the invention of graphical interfaces, we have been trying to apply similar means for communicating with the machine.

Visual Programming Languages are one of many attempts to achieve this. Shapes, colours and text annotations are used to represent computer programs, instead of employing only formatted text. That representation is "first-class" — users are expected to manipulate it directly, instead of changing some underlying text representation. Such programming languages often do not possess human-readable textual representation. There is no widely adopted formal definition. For purposes of this thesis, I use:

> Interactive use of diagrams in the process of programming [9, Chapter 2].

Attempts to create such language were possible only after the invention of graphical interface. The first interactive visual programming language was created by W. Sutherland in 1966 [10]. A simple language based on flowcharts was created in 1969 at

1

Rand [1]. After invention of the first visual programming environments at Xerox Parc in 1970, the main focus of next generations was to improve interface for programmers, while still using textual representation. It is visible in most programming environments, from Smalltalk to Microsoft Visual Studio. No visual programming language ever gained traction comparable to common textual programming languages.

Historical reasons for that are twofold. First of all, although it might be easier for a novice to understand the idea of an algorithm from a diagram, there are no gains for an experienced programmer. For a completely new paradigm to gain traction, current programmers would have to learn it. This would be even more painful than learning a new programming language, as syntax of most mainstream languages with textual representation is similar.

Additionally, most of programming tools are specialised for manipulating text. If a language is text-based, then it can easily be presented, edited, stored and transmitted by multitude of different programs. If, on the other hand, a program is a diagram which can only be rendered by some specialised editor, it becomes much harder to collaborate on it with a group of people, and much harder to share it with other programmers.

1.2 Visual Programming Environments (VPEs)

Since the invention of Smalltalk, programming environments that married textual representation with graphical interface have been becoming more and more popular. Nowadays, powerful integrated development environments (IDEs), such as Visual Studio (https://www.visualstudio.com/) or IntelliJ IDEA (https://www.jetbrains.com/idea/) are commonly used.

Their main advantage over visual programming languages is their versatility — textual representation allowed them to gain popularity without forcing existing programmers to change habits, and allowed them to keep tools that support text. Their chief disadvantage is lack of flexibility. They are blurring lines between the editor and the language — because new language features need new editor features, and so new IDEs dwarf the venerable Emacs (which was often accused of being nothing short of operating system). At the same time, textual representation forces need for keyboard, and doesn't allow experimenting with new input methods. Even mouse is only a secondary mean of interacting with editor. This stands in start contrast with how user interfaces are evolving right now.

Nowadays, touchscreens are getting more and more attention — they are currently the prime interface for smartphones. Thus, most young people are more familiar with them, and not mouse or keyboard. Text is easy to manipulate with keyboard, but hard to manipulate with gestures — visual programming would be far more intuitive for new adepts. Also, programming becoming less of an arcane discipline, and more of a skill needed in every branch of industry. This makes collaboration between programmers and domain experts even more important. Ease of understanding diagrams (and not a specialised programming language) might become a crucial factor in application development, decreasing production time.

1.3 This work

The purpose of this thesis is to address the fundamental problem with Visual Programming Languages: collaboration between programmer using editor presenting diagram representation, and one using standard text editor. This might lead to more widespread use of "programming with diagrams", and better adoption of touch-based input methods in programming, which can potentially revolutionize how programmers work. The main idea is adopting approach from field of functional programming, which makes programs well-structured and concise, while reducing amount of syntactic structures needed to build a program, in order to better visualize it.

First, a functional programming is introduced, along with LISP approach to language design. Then, a dialect of LISP — Clojure — is presented, along with what editor features make working with this language easy. In following chapters we present its visual representation in Jarvis — editor which is the main achievement of this thesis. Jarvis is then described in more details — both how writing programs in it looks like, and how it is implemented. It is then compared with both other VPLs and traditional Clojure editors, to better understand in what it excels.

2 Functional Programming

2.1 What is functional programming?

Functional programming can be defined as:

> Paradigm of programming where fundamental operation is the application of functions to arguments [6].

It means that the program itself is a function which treats its environment as an input and delivers some changes to outside world as a result. The main function is composed of auxiliary functions, which are in turn defined as a composition of more functions. This "functions all the way down" approach (down to a handful of language primitives) leads to very powerful and concise languages. This is caused by better code modularisation in form of higher order functions. It bears resemblance to structured programming being superior to imperative programming. The presence of GOTOs made it impossible to generalise programs (as labels were local and impossible to encapsulate). Functional programming forbids variable mutation (or more generally, side effects). The only possible declarations are static assignments. Mutable states are encapsulated in specialised structures. This facilitates code analysis. Unfortunately, it doesn't come without downsides: it's hard for an average programmer to switch to functional programming.

2.2 How can this be applied to VPL design?

Ideally, a suitable textual representation for VPL would be flexible, powerful and simple. One kind of functional programming languages fits this description especially well. It is LISP (LISt Processor), developed by the Artificial Intelligence group at M.I.T [8].

It is based on Symbolic Expressions (S-expressions) and functions operating on them. Briefly, an S-expression is a list of primitives and S-expressions. A list containing primitives `a, b, c` is represented as `(a, b, c)`. S-functions are functions operating on S-expressions, encoded as S-expressions. LISP is just an interpreter for S-functions. It's remarkable, because encoding language in the same way as its data structures empowers it. For example, one can easily write a function that operates on syntactic forms instead of values. This makes language very powerful, while keeping syntax absolutely minimal. The price for generalised syntax is plenty of similar structures in code and results in code that is hard to read:

```
(defun fibonacci (N)
  "Compute the Nth Fibonacci number."
  (if (or (zerop N) (= N 1))
    1
    (+ (fibonacci (- N 1)) (fibonacci (- N 2)))))
```

3 Clojure

3.1 Introduction

Clojure is a modern functional programming language based on LISP.It's described as:

> Clojure is a dynamic, general-purpose programming language, combining the approachability and interactive development of a scripting language with an efficient and robust infrastructure for multithreaded programming. Clojure is a dialect of Lisp, and shares with Lisp the code-as-data philosophy and a powerful macro system [4].

3.2 Syntax

Lisp's code-as-data philosophy means that Clojure programs are built using its data structures. Generally, Clojure's specification [3] defines Extensible Data Notation (EDN), which is a description of data structures available to Clojure programmers. It includes the following primitives:

- Nil

- Booleans

- Integers

- Floats

- Characters

- Strings

- Symbols

- Keywords

In addition to them, it defines a few compound types, encoded as S-expressions:

- List — `(a b c)`

- Vector — `[a b c]`

- Map — `{a_key a_value b_key b_value}`

- Set — `#{a b c}`

Extensibility is achieved by tags [5]. Any element can be prefixed by a tag, for example:

```
#myapp/Person {:first "Fred" :last "Mertz"}
```

When Clojure's parser encounters such element, it has a couple of choices, one of them being creating a known representation. This, coupled with the fact that only elements readable by Clojure's parser can be tagged, means that the parser that doesn't know a specific tag can still work on a Clojure program containing that tag.

3.3 Semantics

3.3.1 Evaluation

Everything but non-empty lists and symbols evaluate to corresponding data structure. Symbols are resolved, which generally means lookup for a variable corresponding to that symbol. Non-empty lists are interpreted as calls in form of `(operator operands*)`. Whether operands are evaluated, depends on the type of the operator. These include:

- Special forms, which are Clojure's primitives. There are less than 20 in total.

- Macros, which are used to manipulate syntactic abstractions. Operands are passed unevaluated to the macro, and the returned data structure is then evaluated.

If operator is neither a special form nor a macro, the form is interpreted as a function call: both the operator and the operands are evaluated, and the function resulting from the operator's evaluation is called with the results of evaluating the operands as arguments. The returned value is the value of the expression.

3.3.2 Data Structures

Clojure designers opted for integrating more data structures into the syntax to allow encoding complex data structures in a human-readable way. One could easily build a hashmap using a constructor function `(hashmap a b c d)`, although using Clojure's syntax — `{a b c d}` — is more readable and doesn't require eval. Also, it might be worth noting that almost all Clojure data structures are immutable.

3.3.3 Macro Operators

Clojure differs from pure LISP in one more aspect. Its reader recognises some operators and automatically transforms them:

- `@` — dereference: `@some_ref` $\Rightarrow$ `(deref some_ref)` Used to extract value from references and atoms

- `'` — quote: `'foo` $\Rightarrow$ `(quote foo)` Works like LISPs quote, allows stopping evaluation of data structures.

- `` ` `` — syntactic quote: No macro equivalent. Extends quote capabilities to allow escaping (i.e. evaluating only some parts of quoted data structure).

- `~` — syntactic quote escape No macro equivalent Used in conjunction with syntactic quote:

    - `` `~x `` $\Rightarrow$ `x`
    - `` `(x ~y) `` $\Rightarrow$ `(list (quote x) y)`

4 Clojure programming

4.1 Contemporary editors

Despite Lisp syntax being hard for newcomers to understand, Clojure is considered one of the most efficient languages for programmers. Two factors are responsible for that: functional programming and integration with REPL. Virtues of the former were described in the previous chapters, the latter will be discussed in this one. REPL stands for Read-Eval-Print Loop. Much like command line shell, or interactive shells in other languages, it allows the programmer to send the code he's been working on and observe the results directly, without having to execute the whole program. The main difference is what can be read and what can be executed. Due to Lisp's syntax, Clojure's REPL can read, eval and print arbitrary S-expression. Non-Lisp interactive shells need to differentiate between expressions and statements, which can lead to non-intuitive behavior. The second difference is integration with editors. It's common for Clojure programmers to have an editor integration, which allows them to send arbitrary parts of the currently edited file to REPL and observe the results. This results in better understanding of language and less bugs (due to ideally every part of code being tested during writing).

4.2 Editing

As mentioned previously, Clojure is edited as text. Due to the nature of S-expressions, this can prove difficult (keeping so many brackets paired can be troublesome). To mitigate this, most Clojure editors provide special shortcuts which help in editing S-expressions.

Slurp

Slurp adds element to current S-expression by extending parentheses one element to right (forward) or left (backward).

- forward: `(1 2 []) 3` $\Rightarrow$ `(1 2 3 [])`

- backward: `1 ([] 2 3)` $\Rightarrow$ `([] 1 2 3)`

(`[]` is current cursor position.)

Barf

Barf is opposite to slurp — it removes element from current S-expression by moving parentheses left (forward) or right (backward).

- forward: `(1 2 3 [])` $\Rightarrow$ `(1 2 []) 3`

- backward: `([] 1 2 3)` $\Rightarrow$ `1 ([] 2 3)`

Transpose

Transpose re-orders element in current S-expression by swapping element before cursor with one after cursor.

- `(1 [] 2)` $\Rightarrow$ `(2 [] 1)`

Wrap

Wrap simply wraps current element in parentheses.

- `println[]` $\Rightarrow$ `(println[])`

Unwrap

Unwrap removes parentheses from current S-expression.

- `(println[])` $\Rightarrow$ `println[]`

It is mostly used to inline expressions:

- `(println (1 2 3 []))` $\Rightarrow$ `(println 1 2 3[])`

4.2.1 Summary

The pain of editing S-expressions can be largely mitigated by not manipulating text. When S-expressions are structured visually, editing them might be as easy as drawing a new arrow, or moving a box. This would largely remove problems with syntax and allow programmers to focus on the semantics of the program.

5 Jarvis: Visual Clojure editor

5.1 What is its purpose?

Jarvis is a hybrid between Visual Programming Environment and Language. It uses Clojure as its textual representation (so it's not strictly a new language), but presents it in a drastically different way, suitable for manipulation with mouse or gestures. It strives to make Clojure easier for a novice programmer, present the code in a more readable and easier to modify form, provide feedback to the programmer about potential errors and make programming with REPL easier, by providing the ability to directly evaluate expressions. It is implemented as a part of this thesis and it can be downloaded from its webpage https://scoiatael.github.io/Jarvis.

5.2 Programming with Jarvis

Ideally, the coding cycle in Jarvis would be similar to that of other Clojure editors [Figure 1]:
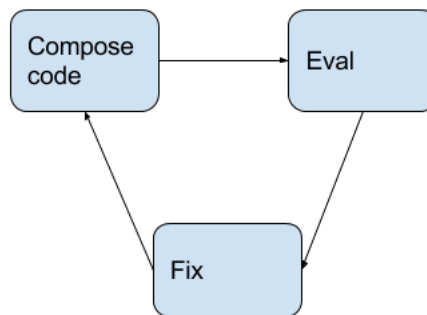
Figure 1: Clojure coding cycle

First, a programmer composes a code from already available functions (either defined in his project, or other libraries). Then the code is sent to the interpreter (inside the editor) and the results are displayed for inspection. Integrating evaluation into the coding cycle enables the programmer to immediately see the results and find the errors. It also enables him to experiment and even find documentation in REPL, instead of using external tools. Then, the errors can be fixed or a different modification can be introduced. This approach poses several challenges. First of all, already defined functions should be listed for the programmer and there should be a possibility of easily adding them to the currently composed code. Secondly, evaluation and displaying of the results should be built into the editor. Furthermore, there should be a way of editing an already evaluated code and re-evaluating it. Last but not least, all this code must be presented in a readable manner, as a diagram of sorts.

5.3 Representation

The most basic functionality of Jarvis is displaying the code as a diagram. Language primitives are shown as a text, while S-expressions are represented as boxes [Figure 2]. Colors are used to signify different types. This creates a representation consisting of embedded boxes, textual nodes as leaves. For example, a function's definition (`defn inc [num] (+ num 1)`) looks like a box with two smaller boxes inside. This has an advantage of presenting S-expressions in a concise way, which is more readable than parentheses.
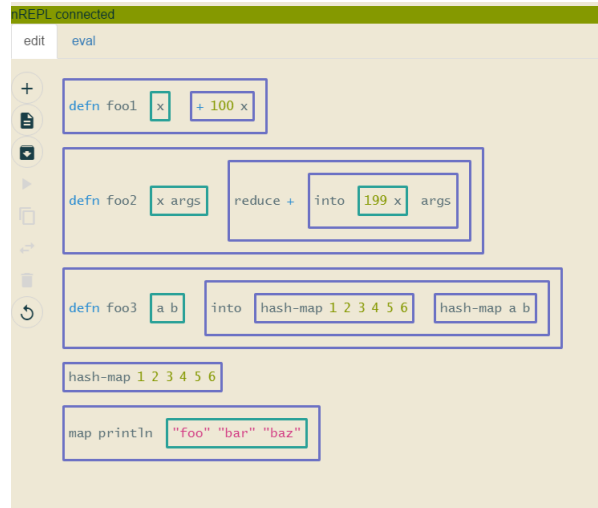
Figure 2: S-Expressions as boxes in Jarvis

5.4 Modification

Modification of a code is implemented as two modes — normal and pasting. It works in a fashion similar to drag & drop — the first click focuses element. Then, the use can either pick it up, or copy it and the editor enters the pasting mode. Then, the second click inserts the element into given place and the editor returns to the normal mode. In the pasting mode possible placement options are shown as buttons [Figure 3]. All clickable elements are highlighted when cursor hovers upon them. Additionally, in the pasting mode, options related to the currently picked up node are available — this enables the user to delete it or evaluate it.

5.5 Evaluation

Code evaluation is solved by providing a second tab in which user can see all previously evaluated expressions. Upon clicking the evaluation button, the node is moved to the evaluation tab and the results of evaluating the expression are displayed [Figure 4].

9

Figure 3: Insert mode in Jarvis

When a user clicks a node in the evaluation tab, it's moved back to the editing tab. This allows the user to rapidly switch between the tabs, running and fixing code.

Figure 4: Evaluation tab in Jarvis

5.6 Installation

Jarvis is available for several platforms from website https://scoiatael.github.io/Jarvis. These include OSX, Windows (both tested), and several Linux distributions (not tested, with underlying build tool still not production-ready) [Figure 5].

In order to evaluate code, Jarvis requires installation of Leiningen (https://leiningen.org/). If installation is not found, error message is shown during startup [Figure 6]. Most of functionality will continue to work, but any attempts to evaluate code will result in an error.
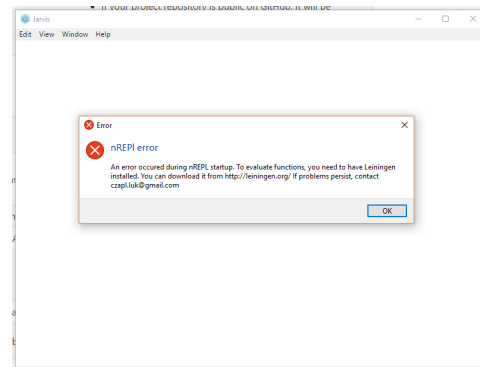
10
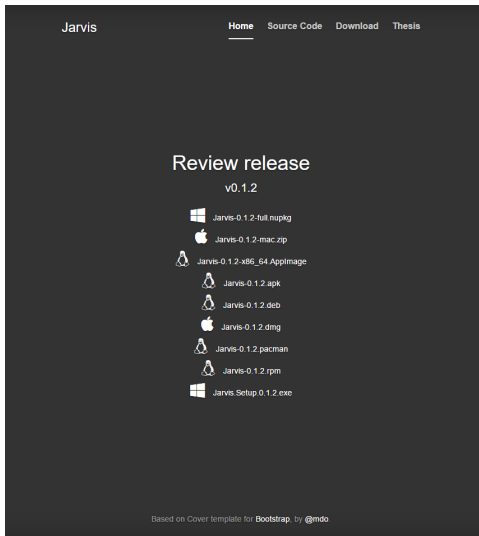
Figure 5: Download page for Jarvis

Figure 6: Leiningen not found error

5.7 Example

In this section we'd like to present writing simple Fibonacci function in Jarvis. After starting Jarvis, user is presented with empty screen. He can open file, or start writing code [Figure 7].
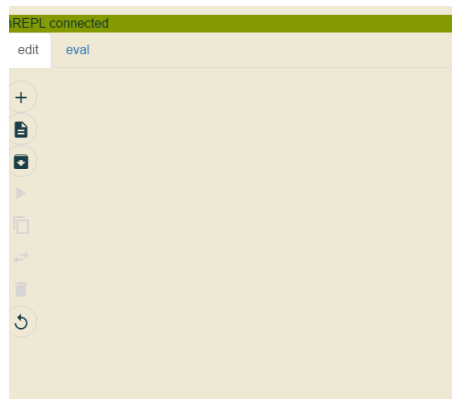

Figure 7: Empty view after opening Jarvis

New elements can be added with Add element button [Figure 8].

It opens a menu with a searchable list of all functions in clojure.core, all functions defined in current namespace, some commonly used constructs, and text box which user can utilize to add custom elements [Figure 9].

We'll use it to add `defn` — function definition. This will create a node which corresponds to head of function definition. It requires also a name of a function and vector of arguments. We'll use text box to add node containing `fib` — function name, `[]` —
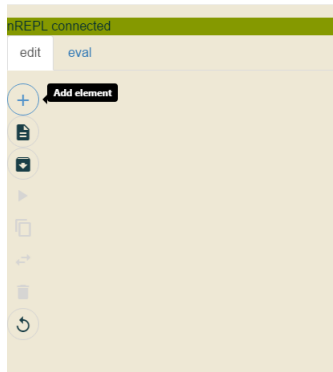
11

Figure 8: Add element button

Figure 9: Add element menu

empty vector of function parameters, and `i` — sole function argument [Figure 10].

Now we can start gluing these parts together. First we focus `i` node with single click [Figure 11], and then use Cut to remove it from its current position and enter pasting mode [Figure 12].
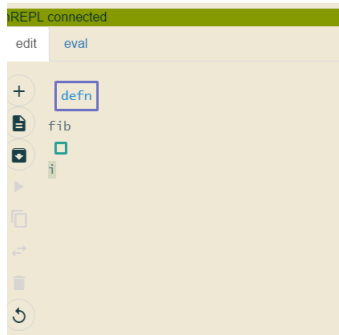

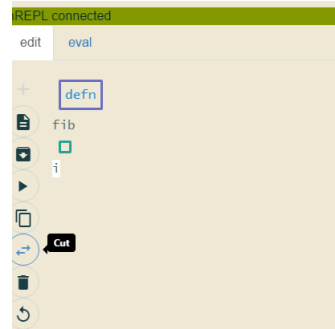Figure 10: Basic parts of `fib` function definition

Figure 11: Cut button

In pasting mode, we can click inside green square (empty vector) to insert `i` inside [Figure 13].

This finishes our vector of parameters [Figure 14].

Using the same mechanism, we finish head of function definition. Then we use Eval button to check for potential errors [Figure 15].

In evaluation window we see that no errors were emitted [Figure 16] — so far our function definition is correct.

Now we click it to return to Edit tab. In there, we add simple `if` element [Figure 17].

After moving it inside `defn` we can again check if no errors were made [Figure 18].
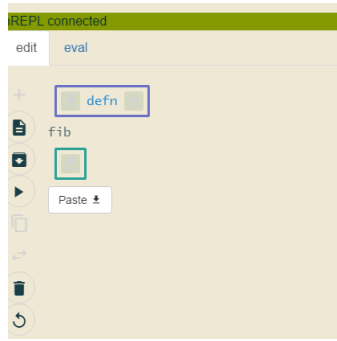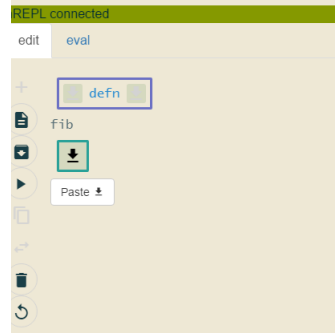
Figure 12: Pasting mode
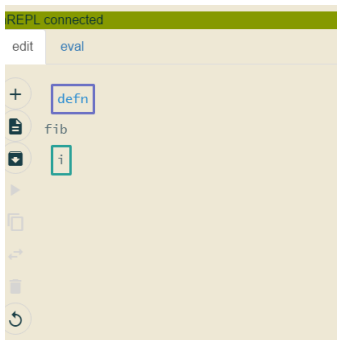

Figure 13: Inserting `i` into vector

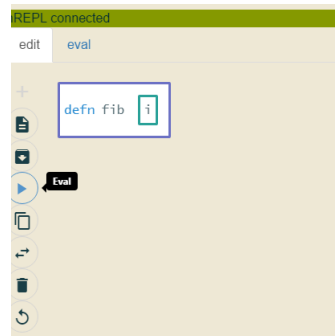Figure 14: Finished vector of parameters

Figure 15: Eval button

Then we proceed to creating second branch of `if` expression — recursive call to `fib` [Figure 19].

It seems our function is done — now we can again evaluate it, and check if it works. We create simple invocation — `(fib 0)`, and evaluate it [Figure 20].

It seems there's an error in our definition — comparison is wrong! We need to change `>` to `<`. This can be done easily by adding new element via text box, cutting it and pasting in place of the other. Then we can check that function works as intended [Figure 21].

5.8 Pros and cons of the current solution

Switching between the normal and "pasting" representation is not very intuitive - the code appears to "jump". An alternative solution might be to show placement locations at all time — this has the disadvantage of requiring a much larger screen in the normal mode. Another possibility could be to open a new window for pasting. For example, the user first picks into which top-level expression he wishes to insert the code, which opens a window with only placement possibilities inside that expression shown.

Moving the code between evaluation and editing tabs might also be surprising to some users — they might think that the code sent to evaluation is somehow lost. Another solution might be to use more tabs — one for editing, one for evaluation, and one for
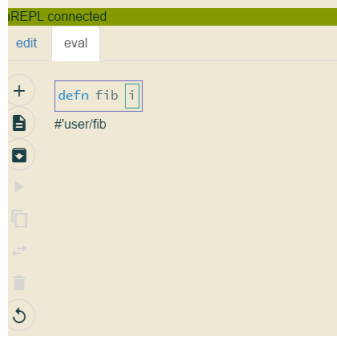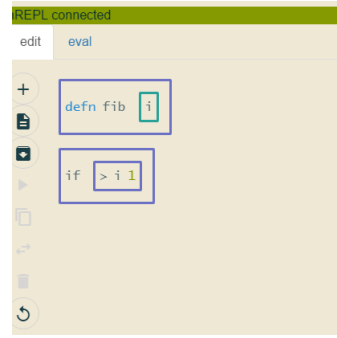
13

Figure 16: Evaluated function definition

Figure 17: Simple `if` node

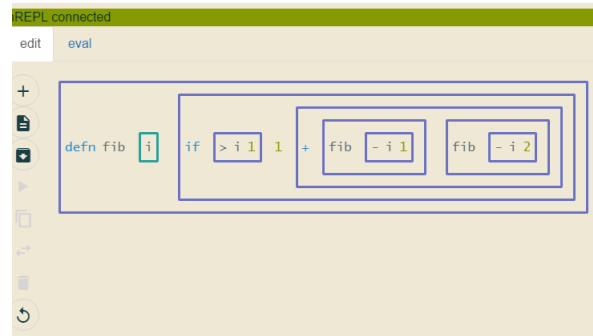Figure 18: Evaluated definition with `if` body

Figure 19: `fib` function ready

the finished code. The user could then either send code to evaluation, or mark it as finished. From the evaluation tab, the code could be copied to either editing or finished (to preserve REPL history) and from finished be moved to editing or evaluation. This has an unfortunate disadvantage of cluttering the interface with unnecessary movement actions (with a possibility of errors and confusing the user).

Another confusing part about the evaluation are side-effects in REPL — if a user evaluates a code, and then moves it to the editing tab, REPL is not cleaned up. All side-effects accumulate, and thus re-evaluating even the same code might yield different results. To solve this, user can force REPL to restart (thus cleaning it up). An alternative solution might involve a wrapper around REPL, which would gather all side-effects of evaluation and clean them up if a node is sent back to editing tab.
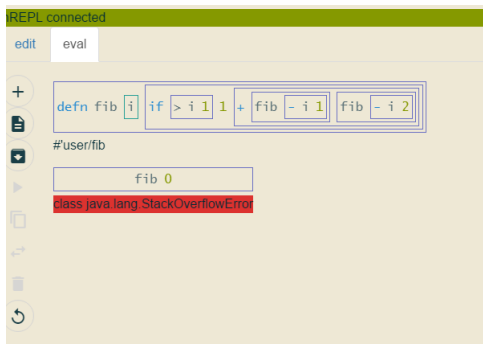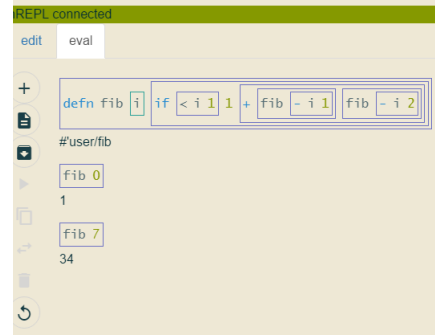
Figure 20: Stack overflow error in testing

Figure 21: Final tests of our Fibonacci function

6 Jarvis implementation

6.1 General Architecture

6.1.1 Why in Clojure?

The biggest advantage of using Clojure for this project are users understanding the codebase. Writing an editor for Clojure in Clojure ensures that if someone uses it, they can also contribute to this project. Potentially, this could mean that even a newcomer might write an extension to Jarvis in Jarvis itself. Because of the dynamic nature of the language, this could even lead to people modifying their editors at run-time! This might make the editor as extensible as Vim or even Emacs.

Secondary gains include a strong suite of existing libraries (which will be presented later) and being a functional language. In this chapter, I'd like to showcase and justify the architectural decisions behind Jarvis.

6.1.2 Single source of truth

State of the whole application is contained in one single Clojure atom, which can be thought of as a database. This way every change in the application maps to a change in this atom. Why is it handy?

- Debugging. If a list of actions leads to a bug showing up, and changes in the state are recorded, the bug can easily be reproduced. Those diffs (changes) can be serialized and sent to the programmer for debugging purposes.

- Understanding the application. Almost every complex application is a state machine under the hood, but very few are explicit about it. With the atom containing the whole state of application, such is the case with Jarvis. It means you can understand the logic behind the whole application by reading the list of state transitions.

- Ease of programming. Having a centralised state means that almost every part of the application can be a function of this state. Views are functions from state to representation that user sees (HTML in this aspect). User action handlers are functions from state and action to new state.

6.1.3 One-way dataflow.

Currently, one of the widespread architectures is Model–View–Controller. Model is responsible for business logic, View for user interaction, and Controller is a bridge between them [Figure 22]. This has numerous disadvantages as an architecture for a big application, because it was meant for small components [7]!
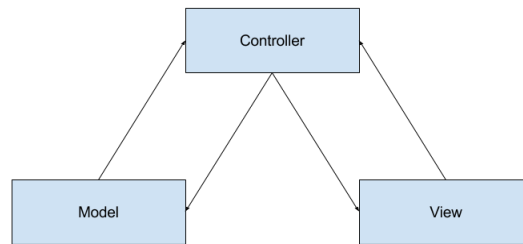

Figure 22: Model–View–Controller

Recently, it has been criticised and an alternative is being searched for. One proposal comes from Facebook: it's Flux architecture. It can be summed up as an unidirectional data flow. In traditional MVC one action might affect multiple models, which in turn might cause more updates in other models. These cascading updates make the application harder to understand. Flux takes an approach that bears resemblance to functional reactive programming. View renders denormalized data which is obtained from the current application state. Ideally, both views and query for data would be declarative. User (and external inputs, like REPL) triggers actions, which are in turn reconciled with current application state — triggering an update of views [Figure 23].

7 Implementation internals

7.1 Technologies:

Jarvis is build on top of technologies that a modern editor, Atom (https://atom.io/), uses. These technologies consist of Chromium rendering engine and Node.js runtime — known as Electron (http://electron.atom.io/). They allow creating cross-platform desktop applications using Web technologies — HTML, JavaScript and CSS. To avoid actually having to write HTML, CSS and Javascript, ClojureScript is used instead. It's Clojure compiled to Javascript. Coupled with extensions allowing for CSS and HTML generation, it allows to write a whole desktop application using the best Web libraries in only one language — Clojure.
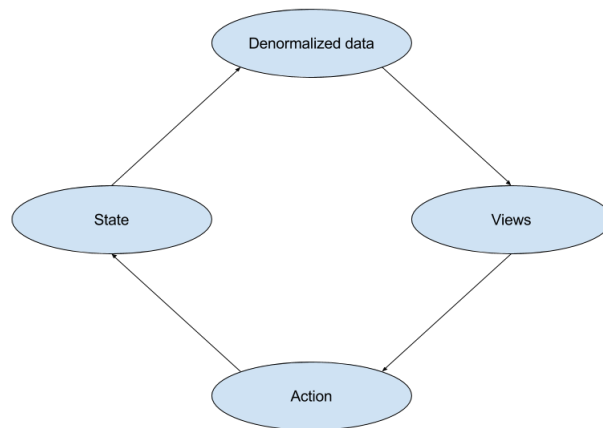
Figure 23: One way data flow

Most of the architectural boilerplate is provided by Re-frame (https://github.com/Day8/re-frame). Efficient, declarative views are provided by Reagent (https://github.com/reagent-project/reagent), which uses Facebooks React (https://facebook.github.io/react/) under the hood. CSS is generated using Garden (https://github.com/noprompt/garden). Additionally, many smaller libraries are included:

- Re-com (https://github.com/Day8/re-com) — GUI components for using with Reagent,

- Figwheel (https://github.com/bhauman/lein-figwheel) — hot code reloader for development,

- Schema (https://github.com/plumatic/schema) — data description and validation library.

7.2 Limitations

In its current form, Jarvis has several limitations. Some of them exist due to the nature of the language, and overcoming others would simply take too much time to implement. One of the latter cases is the simplicity of the checker — although Clojure's basic syntax is just S-expressions, there is a number of predefined functions commonly used — for example `defn` used for function definition. In its most basic form, it accepts a symbol, then a vector of symbols and an S-expression.

```
(defn foo [bar1 bar2] (baz (+ bar1 bar2)))
```

- `foo` — function namespace

- `[bar1 bar2]` — vector of formal arguments

- `(baz (+ bar1 bar2))` — body of function

17

If a part is omitted, runtime error is guaranteed. To improve user experience, the editor could detect errors in such forms. Such checker could also find undefined variables and warn about their usage. It could also check arity of functions. Currently, such functionality in Jarvis is implemented in a proof-of-concept manner — it can detect only the most basic forms of `defn`, `fn`, `def`, `let`. These cover most of Clojure "types" of introducing scope and defining variables and functions.

Another big limitation is the lack of support for data structures other than list and vector. This doesn't limit functionality (as any Clojure program can be written by using their constructors, instead of literals — e.g. `(hash-map 1 2)` instead of `{1 2}`), but doesn't provide good user experience. Same goes for reader forms — for example, lambda-functions syntax (`#(+ 1 2)` instead of `(fn [] (+ 1 2))`) is not parsed correctly. Third limitation is the lack of support for macro operators. Although instead of using `'` (quote) user can user quote function, there is no such equivalent for `` ` `` (syntax quote), `~` (unquote) and `~@` (unquote splice). Additionally, Jarvis has no knowledge of macros — they can be used and defined, but that's hard due to the quoting limitations. Jarvis has also no knowledge of Clojure module system — it currently only lists functions in the current namespace, and these in `clojure.core`. Users would greatly benefit from Jarvis being able to list all of the namespaces available, adding them to the current project, and being able to search and insert variables from any namespace found.

7.3 Possibilities of future extensions

There are several possibilities of extending Jarvis in the future, besides removing the limitations listed in the previous section. The first of them is embedding a Clojure interpreter inside Jarvis, so that it can be directly extended in Clojure — much like Emacs and ELisp work right now. This would allow to add new features and modify existing ones at runtime, allowing users to create extensions (instead of having to add new features directly). Over the years, this has proved a sound strategy for open source editors. Another experiment could be different code representations — current box representation isn't innovative — tree or graph representation might turn out to be more readable. Even switching between them at run-time could be implemented.

Yet another possibility is creating a mobile (targeted for tablets) version of Jarvis. This would require using embedded Clojure instead of running REPL (as it is right now), but would enable a whole new range of devices for which Jarvis would be uniquely suited because of touchscreens. Improving code manipulation is also an option — the current drag & drop interface is not very user-friendly. If combined with more complex checkers, one might detect in which points code can (or should) be added and provide easy visual cues for users.

8 Comparison between Jarvis and other VPLs

8.1 Scratch

Scratch is a VPL designed for educational purposes by the Lifelong Kindergarten Group at the Massachusetts Institute of Technology (MIT). It uses a block-based approach [Figure 24]. Individual elements of the language are represented as pieces of jigsaw puzzles and fit together in different combinations. Drag & drop is a primary method of assembling them, with details tweaked by secondary input options (like paint or sound editor) and text (for numbers and strings).
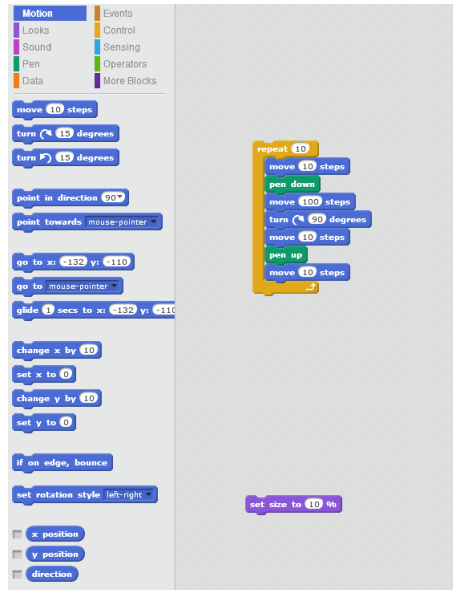
Figure 24: Scratch editing window

8.2 Blueprints

Blueprints Visual Scripting is part of Unreal Engine 4, developed by Epic Games. It aims to help designers and developers help in processes that have traditionally been done by programmers. Interface is node-based — each one represents a part of the language (like a function call), with wires connecting them [Figure 25]. Wire can represent either control or data flow, with both being present in a single script. For example `if` has one control wire and one data wire coming in (execution flow and boolean value), and two control wires coming out (one for `then`, and one for `else` flow). Developers are expected to manipulate nodes and wires with mouse, while underlying implementation for nodes should be provided by programmers as text.

Figure 25: Blueprints

8.3 Features

REPL integration

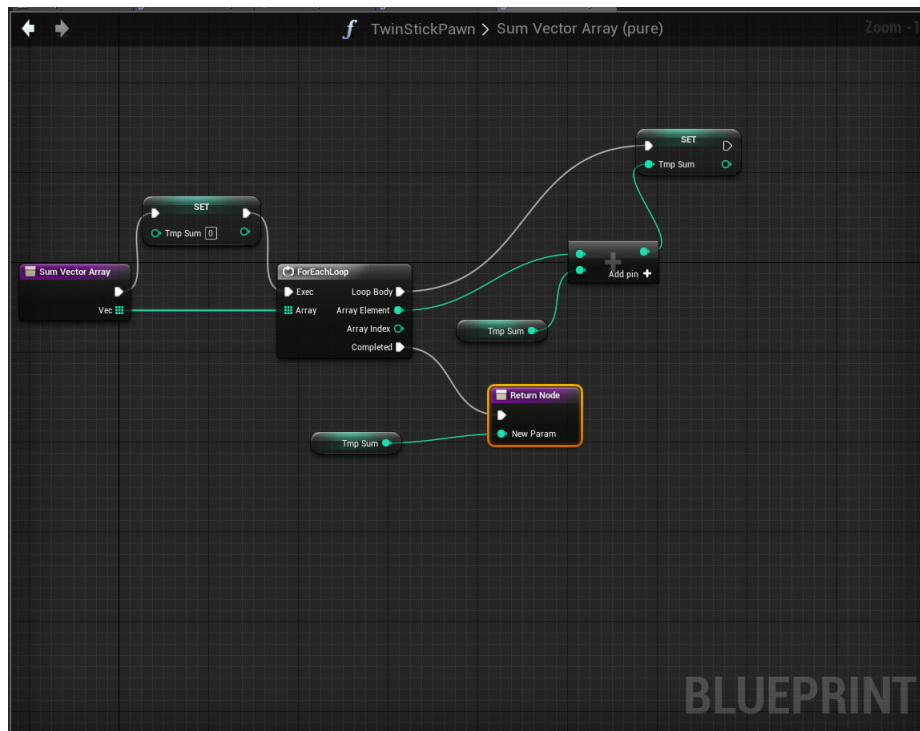Many traditional editors (like Emacs, VIm, or Lighttable) offer possibility to execute parts of program directly from editor, increasing programmers ability to debug program before its fully written. Both Jarvis and Scratch have the same capability, but Blueprints does not possess it. That is caused by Unreal editor model of creating full scenes before running it. Possibly professional programmer would only need to debug scene-by-scene, instead of parts of programmer, but for a novice this proves difficult.

Representation

Jarvis represents program as series of boxes, each representing data structure in Clojure. Scratch uses similar representation — expressions are boxes, and those which can interact (as for example conditional expression, and boolean condition) can interlock, letting programmer intuitive understanding of underlying type system. Blueprints uses nodes (represented by boxes) connected by wires, which represent data and control flows. This offers even more understanding of language to programmer, but is quite tricky to manipulate and get used to.

20

Holes

Both Scratch and Blueprints offer suggestions where some pieces may be missing (by showing respectively empty holes and dangling endings). Of these two Scratch is better. Jarvis does not offer any such insight — that is due to Clojure functions having possibly multiple arities. Such errors are left to be caught during evaluation in REPL.

Comments

Comments increase readers understanding of program. Blueprints allow placing comments on groups of nodes. Scratch and Jarvis have no such capability.

Manual placement

In both Blueprints and Scratch, programmer is expected to place parts of program manually. That creates a potential source of problems — programmer has to worry how his changes would affect placement of whole program. In Blueprints, this problem is partially alleviated by automatic re-wiring — if two nodes were connected, programmer can freely move each of them, and they will still be wired together. Jarvis has fully automatic placement, dictated by structure of underlying Clojure program.

Functions

Ability to group and parameterize code gives programmer very important tool to avoid duplication and make changes easier. All three editors offer such possibility, with least importance given by Scratch (due to its simplicity) and most by Jarvis (due to Clojure being a functional language).

Modules

Building on what functions offer, modules help organize group of functions. Scratch doesn't possess modules, but Jarvis and Blueprints do. Support for them in Jarvis is limited (programmer can import and use functions from other modules, but they are listed in list of currently available functions). Blueprints has no such problems — programmer can easily see and use functions from other modules.

8.4 Using HCI approach

While a list of features helps in understanding what the exact low-level differences between the editors are, in order to better understand what makes one better or worse a different kind of approach is needed. In his work Green presents a framework called "cognitive dimensions of notations" for assessment of cognitive artefacts. It can be used to compare VPLs [2]. In this section, I'll use this framework to compare Jarvis with Scratch and Blueprints.

8.4.1 Abstraction Gradient

Abstraction is:

> Grouping of elements to be treated as one entity, whether just for convenience
> or to change the conceptual structure [2].

This criterium segregates interfaces based on the flexibility of the representation and
the requirements on the programmer for adapting. For example, simple control flow
diagrams that are often used in place of a pseudo-language to define an algorithm, and
state machines, can be categorised as abstraction-hating. Using them, meaning can be
defined quickly, but adding new elements is challenging and requires a lot of explaining.
On the other end of the spectrum are interfaces which require programmers to define
their own abstractions (and thus make each program a unique challenge to a newcomer).
Functional languages are the prime example. Each new function acts as an abstraction.
The program itself is built as an abstraction over more abstractions (akin to a pyramid).
This category is named abstraction-hungry. Middle-ground between them is labeled as
abstraction-tolerant (Pascal or Python are the prime examples).

Scratch is abstraction-tolerant — new abstractions can be defined in form of custom
blocks, but a novice programmer can easily go on without doing so. On the other
hand, both Blueprints and Jarvis are abstraction-hungry. That's caused by both node
(in Blueprints) and box (in Jarvis) being an abstraction over some function. Both
approaches are very similar from the following point of view. A novice programmer has
some traditional primitives (conditionals, loops, arithmetic) available, but is expected
to build a couple of his own abstractions on top of others.

8.4.2 Closeness of Mapping

Solving a problem can be easy in terms of a problem world, but very hard in terms
of a program world. For example, turning car right is easy when you are a driver,
however writing a program that turns car right depends on the abstraction given to you
by both your language and interface of a car programming. Domain specific languages
are designed to solve this problem — to allow programming in terms of the problem
world. VPLs are also pretty good at this, because they are often developed to map to
one domain. Scratch has a clearly defined problem world (moving objects on screen)
and its mapping is very close, with (almost) every piece of puzzle representing some
action in the problem world. For general-purpose languages, that evaluation is much
harder. It requires thinking about a very simple problem world and considering how
many language primitives are needed to solve a problem there.

Here, Jarvis seems to have the upper-hand. Blueprints require additional wiring of
control flow, and many nodes associated with manipulation of local variables.

8.4.3 Consistency

A notion of language being consistent or not is hard to define objectively. Inconsistencies
for end-users might arise as a result of a very consistent approach from the language

designer. One might argue that the simplest way to avoid such inconsistencies is making the language as simple as possible. The comparison of Scratch, Blueprints and Jarvis does support that claim: all three have a much simpler structure than textual languages and no inconsistencies are present. Jarvis (because of underlying Clojure) seems to be the best — with only a handful of primitives, consistency of everything else is up to the programmer. Scratch has more language elements, which are grouped by type (represented by shape) and category (represented by colour). Blueprints is built on top of primitives provided by the programmer, with only a representation being defined.

8.4.4 Diffuseness/Terseness

This criterium is hard to measure independently of Closeness-of-Mapping, as language closer to the problem world will require less primitives to solve it. Another problem is that the ideal language cannot be either too diffuse or too terse. If too terse, the programmer will have a hard time finding bugs, as they may be virtually invisible. If too diffuse, programmer has to read and memorise a lot to understand what's happening. For VPLs this is especially important, as the total amount of screen space needed to represent a program should be taken into account. Scratch is by far the most terse of the three compared. Jarvis and Blueprints require much more screen space — with Blueprints using different tabs for each function and supporting multiple levels of zoom to allow the programmer to either inspect the details or focus on a bigger picture. Jarvis supports two views — one for program parts being currently edited (which takes more space, to assist in editing) and one for parts that have already been executed (which takes less space, to make reading easier).

8.4.5 Error-proneness

Generally, VPLs are much less prone to syntax errors than traditional textual languages. Their representation makes it virtually impossible to make a syntax error. Jarvis uses boxes to save the programmer from trouble with parentheses, the Scratch jigsaw puzzle makes it impossible to make an error, and Blueprints will not allow the programmer to wire the program in a wrong way.

8.4.6 Hard Mental Operations

Conditionals and boolean arithmetic are a prime example of a notation gone wrong. It's easy to express an easy-to-grasp idea in an incomprehensible way. Surprisingly, [2] shows that although diagrams look like a good way to avoid such problems, they often fail to do so. Even if shown as a diagram, conditionals imbued in more conditionals resemble a visual exercise on a formal logic course, not something easy to grasp. Scratch suffers the most from this, as it uses imperative programming as its base. Blueprints (with Object Oriented programming) are a bit better, and Jarvis (with functional programming) fares best. That is solely caused by how often that kind of hard mental operations happen in underlying programming thinking patterns and not by representation.

8.4.7 Hidden Dependencies

Hidden dependencies are

> Relationships between two components such that one of them is dependent on the other, but that the dependency is not fully visible [2].

They are caused by side-effects (such as assigning a value to a shared variable). Generally, VPLs fare better than textual languages when it comes to low-level (local) dependencies — simply because they have more means to visualise that dependency. When it comes to high-level dependencies, they don't seem to fare any better. So is the case with Scratch and Blueprints, with Jarvis showing more promise. Both Scratch and Blueprints have global variables, and no good support for showing where they are modified. Jarvis (thanks to Clojure), has a better model for handling side-effects, but no means of visualising it.

8.4.8 Premature Commitment

When solving problems, the programmer should worry about the problem world. Many programming languages pose many requirements on the structure of the program, making working with them tiring, as programmer has to cope with a lot of additional guessing ahead. This criterion measures how much additional guesswork is associated with solving problems in the given language. Generally, VPLs using 2D spatial representations tend to pose less difficulties than textual languages, but are not without problems themselves. For box-and-wire languages, the placement of boxes with a minimum number of wires crossing requires a guess-ahead. Such is the case with Blueprints. For Scratch, the problem is with placing puzzles to group them and leaving enough space for subproblems. The Jarvis' problem is similar to the problems of textual languages — the order of evaluation. Parts sent to REPL first are shown first and there's no way of re-ordering them (besides moving them back to editing and evaluating later, which places them at the end). This makes leaving holes for subproblems impossible, thus the whole program resembles a tree flattened in a prefix fashion. Imposing some structuring on such list is challenging, to say the least.

8.4.9 Progressive Evaluation

Some languages forbid running programs which are not complete. Other — mostly dynamic interpreted languages allow running parts of a program. Ideally, the programmer can write a program by interacting with REPL and saving parts which are working. Such is the case with Jarvis. The program is constructed by running small expressions and composing them to solve bigger problems. Scratch also allows running expressions, but results are shown by changing the state of stage — to see them again, you need to re-run the expression. Blueprints doesn't allow running any expressions because everything is wired to a global stage. You can only run the whole game and observe what changes were introduced by your code modification.

8.4.10 Role-expressiveness

Generally, programmers agree that some languages are harder to read than others. Unfortunately, there's no consensus on which of them. C-programmer might argue that LISP is an unreadable mess of parentheses, while his adversary would say that the bloated syntax of C makes understanding even simplest programs challenging. To avoid this argument, one can consider how easy it is to understand what a given expression is for.

> Role-expressiveness is presumably enhanced by the use of meaningful identifiers, by well-structured modularity, by the use of secondary notation to signal functionally-related groupings, and by the presence of 'beacons' that signify certain highly diagnostic code structures [2].

Scratch offers no means of modularity besides defining functions and no secondary notation besides the placement of blocks. There are also no diagnostic code structures. Blueprints allows to add comments on a group of nodes. Jarvis, on the other hand, has no support for comments, but allows using Clojure pre- and postconditions to annotate functions with extra information about arguments and returned values. This can be additionally extended by using proper Clojure libraries (which can add such features as types of functions).

8.4.11 Secondary Notation and Escape from Formalism

As discussed before, main secondary notation for VPLs is the spatial placement of items. That is only possible in Scratch, with Blueprints supporting additionally comments. In Jarvis, the programmer has no secondary means of conveying information. Indenting and manual placement of items are impossible, comments are not supported. The programmer is entirely dependent on language features.

8.4.12 Viscosity: resistance to local change

Programmers tend to make mistakes. Some are easier to fix than others, which often depends on the language. The biggest issue of Scratch is the order in which items have to be moved (generally, moving one item moves the rest below it). This sometimes imposes a big amount of changes needed to move just one block. Blueprints has a problem with wiring — although wires are placed automatically, changing a block can require moving a lot of other blocks to make the program readable again. Jarvis has a problem with shuffling blocks.

8.4.13 Visibility and Juxtaposability

This criterion measures how easy it is to just see what the program does, without much mental work. This is tightly coupled with the ability to see different parts of the program at the same time.

Neither Scratch, Blueprints, nor Jarvis offer such a thing. Overall, all three VPLs make programmer's life hard in this aspect, which is especially shameful for Jarvis, because most Clojure editors allow instantly looking up the definition of any symbol. Code navigator would also be a big improvement here.

9 Comparison between Jarvis and mainstream Clojure editors

In this section we'll compare Jarvis with other editors commonly used to write Clojure programs: VIm, Emacs, and Lighttable.

9.1 VIm

VIm (http://www.vim.org/) is small but easily extensible editor, first released in 1991. Clojure editing is improved by several plugins — some of them provide REPL support, other offer better means of editing S-expressions.

9.2 Emacs

Emacs (https://www.gnu.org/software/emacs/) is surprisingly not a text editor — it is LISP interpreter, with text editing being only one of its functions. As with VIm, Clojure editing is provided by many plugins, which often build on top of one another.

9.3 Lighttable

Lighttable (http://lighttable.com/) is a "next generation code editor". It tightly integrates with language edited, providing means to easily run and watch program being executed from editor. One plugin provides integration of Clojure into Lighttable.

REPL integrations

All of compared editors possibility to execute parts of program directly from editor.

S-expressions editing

All of compared editors offer possibility to manipulate S-expressions (using amongst other techniques described in [Section 4.2]).

Project awareness

Emacs, Lighttable and VIm support multi-file projects, by providing means of easily switching between files and cross-referencing functions. Jarvis focuses and editing one file, and offers means of switching easily (besides opening new files, which leaves current file altogether — switching quickly is impossible).

Refactoring

All three textual editors offer support for refactoring (renaming, extracting function definitions, inlining functions). Jarvis has no such capability, although programmer should be able to add it quite easily (via REPL integration, similar to how it is implemented in VIm and Emacs).

Syntax check

All textual editors offer ability to detect and present possible errors, in varying degrees — with most basic being detection of misplaced parentheses. In this aspect Jarvis proves superior — programmer simply cannot make such mistake. Jarvis also tries to analyze code and show possible semantic errors with help of REPL.

Autocomplete

Most textual editors currently in use have ability to autocomplete identifiers (to avoid typing full names), either by scanning all words in current document or more sophisticated means. Jarvis connects to REPL and offers programmer list of functions available currently to him.

Syntax highlighting

Color-coding different language parts (identifiers, brackets) helps programmer in understand program structure. VIm, Emacs and Lighttable offer this, but Jarvis offers more — it visualizes program as diagram, which adds another dimension to representation. This might prove especially important to novices.

Automatic indentation

VPLs often place nodes automatically — textual editors try to do the same by automatically indenting the code. It removes the need for programmer to structure the code by himself. It is present in Emacs, VIm and Lighttable, with Jarvis offering similar capability by placing all boxes automatically.

10 Summary

Representing Clojure program — a language based on LISPs idea of S-expressions and data–as–code — as a diagram proves to be a successful experiment. Jarvis is a new kind of Visual Programming Environment, one that allows having all pros of "programming with diagrams", while avoiding downsides traditionally associated with Visual Programming Languages. This approach, if developed, could lead to programming environments using entirely different means for communication with programmer then keyboard and mouse — for example touchscreens.

Compared to traditional Clojure programming environments (Emacs, VIm, Lighttable), it lacks features associated with multi-file projects, but proves superior in representation and error avoidance. Compared to other VPLs (Scratch, Blueprints), it lacks for features improving user experience, but generally is on the same level of usability.

In addition, there are many potential development ideas for Jarvis, which could lead to it becoming a professional programming environment.

References

[1] J. F. Heafner Ellis, T. O. and W. L. Sibley. The grail project: An experiment in man-machine communications.

[2] T. R. G. Green and M. Petre. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. Journal of visual languages and computing, 7:131–174, 1996.

[3] Rich Hickey. Clojure reader specification. [Online; accessed September 5, 2016].

[4] Rich Hickey. Clojure website. [Online; accessed September 5, 2016].

[5] Rich Hickey. Edn specification. [Online; accessed September 5, 2016].

[6] J. Hughes. Why functional programming matters. Comput. J., 32(2):98–107, April 1989.

[7] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. J. Object Oriented Program., 1(3):26–49, August 1988.

[8] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. Commun. ACM, 3(4):184–195, April 1960.

[9] Jeffrey Vernon Nickerson. Visual programming. 1995.

[10] William Robert Sutherland. The on-line graphical specification of computer procedures. Massachusetts Institute of Technology, 1966.