GRASP: An Extensible Tactile Interface for Editing S-expressions

Panicz Maciej Godek
godek.maciek@gmail.com

ABSTRACT

GRASP is a Scheme-based extensible computational environment designed to work with S-expressions on touch screens. It features a powerful extension mechanism as well as a subsystem for handling gesture-based input. It is implemented in Kawa Scheme, and can be compiled as an Android application as well as run on a desktop windowing environment and inside of terminal emulators.

GRASP is still a work-in-progress application, so the purpose of the demo is:

(1) to show the current state of the application;
(2) to convey the ultimate vision behind GRASP;
(3) to present the development plan and methodology, and optionally:
(4) to describe the hitherto history of the development.

The presentation of GRASP in this paper is written as if all of its features were already implemented. The omissions are presented in a separate section.

CCS CONCEPTS

• Software and its engineering → Integrated and visual development environments; • Human-centered computing → Visualization toolkits; Ubiquitous and mobile computing systems and tools; • Computing methodologies → Graphics input devices.

KEYWORDS

visual programming, touchscreen-based editing, interactive programming, structual editing

ACM Reference Format:
Panicz Maciej Godek. 2023. GRASP: An Extensible Tactile Interface for Editing S-expressions. In Proceedings of the 16th European Lisp Symposium (ELS'23). ACM, New York, NY, USA, 4 pages. https://doi.org/10.5281/zenodo.7816633

1 THE CONCEPT OF GRASP

GRASP[1] is a tactile-first structural editor for S-expressions. Its design is based on representing S-expressions as nestable boxes. The boxes are rendered so that their left and right edge resemble – respectively – opening and closing parentheses.

When displayed in a terminal, a Lisp program edited in GRASP might look like this[2]:

```
┌        ┌     ┐                       ┐
│ define │ ! n │                       │
│        └     ┘                       │
│ ❝ ──────────────────────────────── • │
│ │ Computes the product 1*...*n.    │ │
│ │ It represents the number of per- │ │
│ │ mutations of an n-element set.   │ │
│ • ──────────────────────────────── ❞ │
│   ┌    ┌        ┐                 ┐  │
│   │ if │ <= n 0 │                 │  │
│   │    └        ┘                 │  │
│   │                               │  │
│   │       1                       │  │
│   │                               │  │
│   │       ┌     ┌   ┌       ┐ ┐ ┐ │  │
│   │       │ * n │ ! │ - n 1 │ │ │ │  │
└   └       └     └   └       ┘ ┘ ┘ ┘  ┘

┌      ┌             ┐          ┐
│ e.g. │ factorial 5 │ ===> 120 │
└      └             ┘          ┘
```

which corresponds to the following program text:

```
(define (! n)
"Computes the product 1*...*n.
It represents the number of per-
mutations of an n-element set."
  (if (<= n 0)
      1
      (* n (! (- n 1)))))
(e.g. (factorial 5) ===> 120)
```

The left and right parentheses play different roles in tactile editing: the left parenthesis is used for moving (if pressed once) or copying (if pressed twice) an expression, whereas the right parenthesis is used for resizing an expression.

An expression which is currently being moved can be deleted by throwing it off the surface quickly. Likewise, moving a finger quickly while the expression is being resized causes the box to be spliced into its parent (this feature is sometimes referred to as "pulling-the-rug splicing").

In addition to boxes, GRASP offers four other types of objects: atoms, texts, extensions and comments.

[1]GRASP is an open-source project hosted at https://github.com/panicz/grasp

[2]Some recordings presenting various prototypes of GRASP can be watched on the author's youtube channel: https://www.youtube.com/channel/UCt4u6WQDy2yjXz6eXCcyijQ

Atoms are things like symbols, numbers, characters or Boolean values in Lisp. They support touch gestures in a similar way as the left parenthesis of a box: single touch causes them to be dragged, whereas double touch causes their copy to be dragged.

The text type corresponds to strings. They are displayed inside boxes with quotation marks on their corners. The roles of the quotation marks are analogous to the left and the right parenthesis: the left one can be used to move the text within the expression tree, remove it or copy, while the right one can be used to change the shape of a text.

Comments in the Scheme programming language come in three flavors, all of which are supported by GRASP:

- line comments, which span until the end of a given line;
- block comments, which are similar to text;
- expression comments, which comment out a single expression.

Comments are invisible to the operations on the document, such as `car` or `cdr`. Other than that, line and block comments are similar to text.

The last type of objects supported by the editor are extensions. The list of extensions is open-ended. Expressions are sometimes referred to as "magic boxes", because they are boxes which define their own rules of interaction.

A simple example of an extension is a button. If it is loaded, the expression

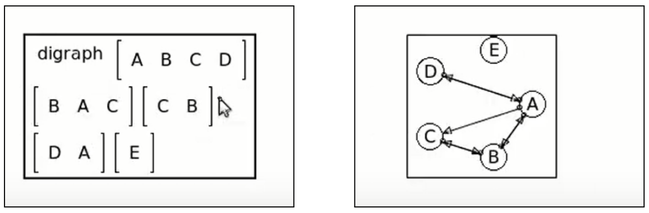
```
(Button label: "Press me"
        action: (lambda () (WARN "button pressed")))
```

can be rendered as a button, and responds to touch events with the invocation of its action callback.

The terminal client of GRASP would display it in the following manner:

```
┌─────────┐
│Press me!│
└─────────┘
```

A more advanced extension – coming from an earlier prototype of GRASP – allows to display graphs represented in the form of neighbour list as an actual graph:



```
digraph [A B C D]
[B A C] [C B]
[D A] [E]
```

Extensions are meant to be user-definable, but the exact API for defining them is subject to an ongoing research.

Some desired extensions for GRASP include:

- a drawing editor
- a graph visualizer/editor
- a visual evaluator
- a function plotter

and many others.

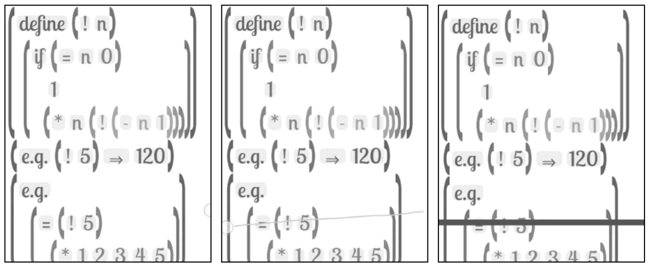
1.1 Gesture-based input

Since devices with touch screens often lack a proper keyboard, and usually display regrettable keyboard substitutes on their screens as needed, GRASP attempts to find a more ergonomic alternative.

One idea is gesture-based input: the user draws a shape on the screen, and if the shape is recognized, an appropriate action is performed.

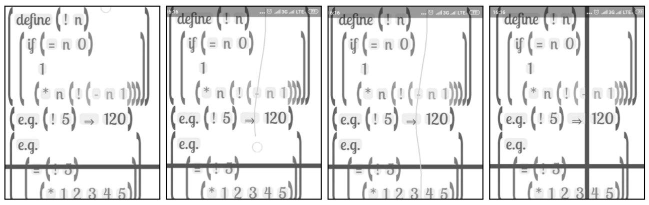
By default, the following shapes are recognized:

- horizontal line, which splits the panes it's drawn over vertically into halves (similar to `C-x 2` in Emacs)

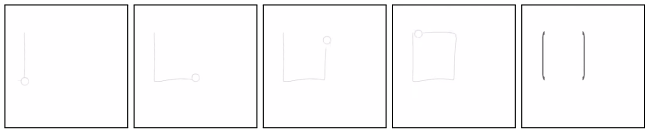


```
(define (! n)
  (if (= n 0)
    1
    (* n (! (- n 1)))))
(e.g. (! 5) ⇒ 120)
(e.g.
  (= (! 5)
    (* 1 2 3 4 5)
```

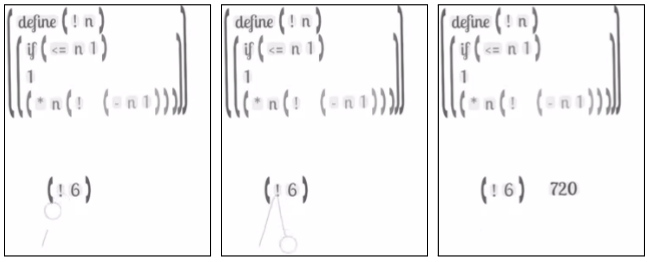
- vertical line, which splits the panes below horizontally into halves (similar to `C-x 3` in Emacs)



- box gesture, which creates a new box in the document it's drawn over



- wedge symbol, which causes the expression below its blade to be evaluated (similar to `C-x C-e` in Emacs' Lisp interaction modes)



```
(define (! n)
  (if (<= n 1)
    1
    (* n (! (- n 1)))))

(! 6)  720
```

Since many touchscreen-equipped devices also feature accelerometers, GRASP also lets define motion-based edit operations – for example, shaking a device might result in reindenting the source code.

1.2 Keyboard input

Even though GRASP focuses on tactile editing and on mobile devices, a lot of effort has been put into making it a pleasant keyboard editing experience.

GRASP features a flexible key binding mechanism, which unites the input systems from its target environments (Android, terminal and windowing systems).

By default, it provides the "Common User Access" keyboard bindings (ctrl-z for undo, ctrl-c for copy etc.) and it allows users to use keyboard arrows to navigate cursor over the active document.

Keyboard editing is context-sensitive, so for example pressing the `#\[` key creates a new box, unless the cursor is located on a text element, in which case the `#\[` character is inserted verbatim into text.

Also, extensions are free to interpret most of the pressed characters as they please.

2 STRUCTURAL EDITING

The documents in GRASP are considered mutable, and the editing of a document occurs by means of mutating their tree structure.

However, all these mutations are inter-mediated by explicit Edit operations. Each such operation has its inverse, which on one hand is used to implement the undo mechanism, and on the other – can be perceived as an interesting "document editing assembly language".

At the moment of writing this text, the language consists of the following (invertible) operations:

```
(Move from: Cursor to: Cursor with-shift: int)
(Insert element: (either pair HeadTailSeparator)
        at: Cursor)
(Remove element: (either pair HeadTailSeparator)
        at: Cursor with-shift: int := 0)
(ResizeBox at: Cursor := (the-cursor)
           from: Extent
           to: Extent
           with-anchor: real)
(InsertCharacter list: (list-of char)
                 after: Cursor := (the-cursor)
                 into: pair := (the-document))
(RemoveCharacter list: (list-of char)
                 before: Cursor := (the-cursor))
(SplitElement with: Space
              at: Cursor := (the-cursor))
(MergeElements removing: Space
               at: Cursor := (the-cursor))
```

Some of these operations are pairwise inverse (e.g. `Insert` and `Remove` or `SplitElement` and `MergeElements`), while others are self-inverse (e.g. `Move` or `ResizeBox`).

More details can be found in the source code of GRASP.

It is imaginable that some future version of GRASP could observe the actions performed by user and the structure of the document, and suggest certain operations based on previous actions (resembling Excel's auto-fill feature).

3 CURRENT PROGRESS

Although this paper could leave a different impression, at the moment of writing (February 2023) GRASP isn't yet a usable application, as:

- it doesn't let users open or save files
- it doesn't let users split or scroll the screen
- it doesn't let users evaluate expressions
- it doesn't support the basic gestures
- the extension mechanism isn't available

In certain areas, it also seems to have similar shortcomings:

- it doesn't support displaying nor editing comments
- although it should display improper lists correctly, editing them has not been tested well

Fortunately, there's still some time before European Lisp Symposium, which takes place late in April. Currently, the author envisions two milestones for the project:

(1) to reach the point that would let GRASP be used for developing itself
(2) to support extension mechanism and focus on the development of particular extensions

The author believes that reaching milestone 1 before ELS might be possible. A more detailed plan is the following:

- support for keyboard editing (mostly done)
- support for displaying and editing comments (they are already handled by parser)
- support for vertical keyboard movement (currently works somewhat but is a kludge)
- support for loading and saving files
- support for screen splitting and scrolling
- support for syntactic extensions provided by Kawa that are used in GRASP
- tests and bug fixes

4 RELATED WORK

The strongest source of inspiration for GRASP has been Emacs[14], and the Scheme interaction mode provided by the Geiser package. One motivation for the development of GRASP was the desire to share experience of Lisp interaction mode outside the world of Emacs, with possible improvements. (Some fundamental shortcomings of Emacs were pointed out with the announcement of Project Mage in the January of 2023[11].)

The desire to add interactive visual extensions was born when the author attempted to extend the idea of "evident programming" to the domain of computational geometry and graph algorithms.

However, the same idea was independently conceived by Leif Andersen, who implemented it in Dr Racket, and then created a browser-based IDE called visr.pl (for Clojure). Leif also provided a very good explanation of the idea in a youtube video [1].

Interactive visual syntax is also a key feature of the Polytope editor developed by Elliot Evans. Polytope is a dedicated editor for JavaScript [8].

There are many similarities between GRASP and the Boxer environment developed at MIT in the 1980s by Andrea DiSessa and Harold Abelson [4]. Recently there have been efforts to resurrect Boxer within the Boxer Sunrise project run by Antranig Basman and Steven Ghitens [5]. However, building the project requires LispWorks, and pre-built snapshots are only released for MacOS X. Also, despite being written in Lisp, Boxer itself is not a Lisp interpreter.

There used to be a Boxer-inspired "integrated Scheme programming environment" called Bochser, developed by Michael Eisneberg in the 1980s at MIT[3].

Despite similarities, Bochser is a very different system than GRASP, and with very different goals.

Eisenberg's thesis contains a reference to another thesis, which presents Franklyn Turbak's "visual and manipulable model for Scheme programs" called GRASP[15]. It has even less in common with the system presented here than Bochser.

There are other interesting experiments in the area of representing programs. One example is the Fructure editor developed by Andrew Blinn for the Racket programming language (the editor itself is implemented in a purely functional way, using Racket's "big-bang" library)[6].

Another is OrenoLisp designed by Yasuyuki Maeda with the purpose of artistic live music coding[12].

There's a fun representation of ClojureScript programs as nested circles invented by Ella Hoeppner for her Vlojure editor[10].

Katie Bell created a browser-based structural editor for Python called SplootCode[2].

A lot of work concerning data visualization has been happening around the Smalltalk distribution called Pharo, and in particular its spin-off called Glamorous Toolkit, developed by Tudor Girba and his associates[9].

There's also a Visual Studio Code plug-in called "Debug Visualizer" developed by Henning Dieterichs[7]. It lets visualize various data structures during the execution of programs, and is available for the majority of mainstream programming languages.

While the scene of structural editing tools seems to be flourishing, the same cannot be said about development tools for mobile devices - most of existing tools seem to be shrinked versions of PC-based development environments and require external keyboard for comfortable work.

The only tool which stands out from this crowd that the author of this work knows about is MobileCode (formerly medc) developed by Mark Mendell[13], which is a vim-inspired touchscreen-based editor for C-like languages, capable of collapsing procedures and blocks of code.

ACKNOWLEDGEMENTS

The author would like to thank Shriram Krishnamurthi for reading the first draft of this work. He is also grateful to Andrew Blinn for his spiritual support throughout the whole development process, and the whole online scene of people interested in making programming a better experience for the future generations.

The terminal version of GRASP owes the use of Unicode box-drawing characters (rather than slashes and backslashes) to Job van der Zwan. After seeing the first prototype, Manuel Simoni recommended not to grow the boxes vertically with the increasing nesting level.

REFERENCES

[1] Leif Andersen, Michael Ballantyne, Mathias Felleisen, Adding Interactive Visual Syntax to Textual Code, Proceedings of the ACM on Programming Languages, Volume 4, Issue OOPSLA, Article No.: 222, pp 128, https://doi.org/10.1145/3428290, presentation: https://www.youtube.com/watch?v=8htgAxJuK5c defense talk: https://www.youtube.com/watch?v=l0GfMs82PvU online IDE: https://visr.pl
[2] Katie Bell, SplootCode, https://splootcode.io
[3] Michael Eisenberg, Bochser: An Integrated Scheme Programming System, MIT 1985, https://boxer-project.github.io/boxer-literature/theses/Bochser,AnIntegratedSchemeProgrammingSystem(Eisenberg,MITMSc,1985).pdf
[4] Andrea DiSessa, Harold Abelson, Boxer: A Reconstructible Computational Medium, MIT 1986, https://web.media.mit.edu/~mres/papers/boxer.pdf
[5] Antranig Basman, Steven Ghitens, Boxer Sunrise Project https://github.com/boxer-project/boxer-sunrise
[6] Andrew Blinn, Fructure: A Structure Editing Engine in Racket source code: https://github.com/disconcision/fructure 2019 RacketCon presentation: https://www.youtube.com/watch?v=CnbVCNIh1NA
[7] Henning Dieterichs, Debug Visualizer for Visual Studio Code https://marketplace.visualstudio.com/items?itemName=hediet.debug-visualizer
[8] Elliot Evans, Polytope, https://elliot.website/editor/
[9] Tudor Girba, Glamorous Toolkit, https://gtoolkit.com
[10] Ella Hoeppner, Vlojure: A New Way to Write Clojure, presentation: https://www.youtube.com/watch?v=1OcAUhe3E1E online IDE: https://vlojure.io/
[11] Dmitrii Korobeinikov, Emacs is Not Enough, Project Mage, 2023, https://project-mage.org/emacs-is-not-enough
[12] Yasuyuki Maeda, OrenoLisp, https://www.youtube.com/watch?v=RuU0HI-paik
[13] Mark Mendell, MEDC project website: https://medc.mark.dev/ presentation: https://vimeo.com/641790697
[14] Richard Stallman, EMACS: The Extensible, Customizable Display Editor, 1981, https://www.gnu.org/software/emacs/emacs-paper.html
[15] Franklyn Turbak, GRASP: A Visible and Manipulable Model for Procedural Programs, MIT 1986 https://cs.wellesley.edu/~fturbak/pubs/turbak-masters-thesis.pdf