GRASP: An Extensible Tactile Interface for Editing S-expressions

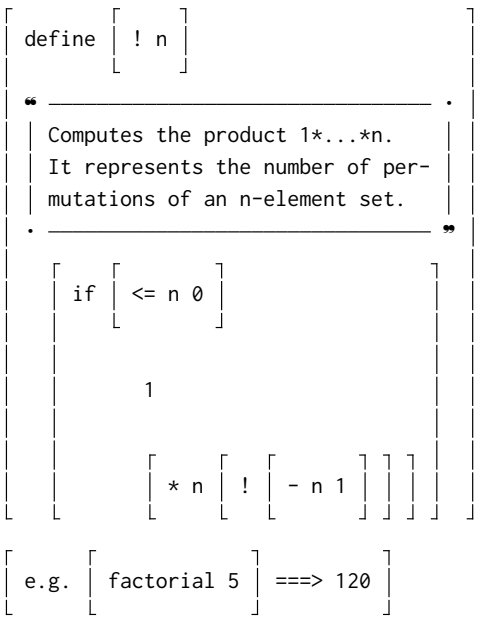
Panicz Maciej Godek
godek.maciek@gmail.com

ACM Reference Format:
Panicz Maciej Godek. 2023. GRASP: An Extensible Tactile Interface for Editing S-expressions. In Proceedings of European Lisp Symposium (Conference acronym 'ELS). ACM, New York, NY, USA, 7 pages. https://doi.org/XXXXXXX.XXXXXXX

1 THE CONCEPT OF GRASP

GRASP 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:



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.

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

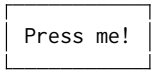
```
(Button label: "Press me"
        action: (lambda () (WARN "button pressed")))
```

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

Conference acronym 'ELS, April 24–25, 2023, Amsterdam, NL

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

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



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 pane it's drawn over vertically into two smaller panes (similar to `C-x 2` in Emacs)
- vertical line, which splits the pane below horizontally into two smaller panes (similar to `C-x 3` in Emacs)
- a box gesture, which creates a new box in the document it's drawn over
- an underscore gesture, which creates a new atom in the document it's drawn over
- a wedge symbol, which causes the expression below its blade to be evaluated (similar to `C-x C-e` in Emacs' Lisp interaction modes)

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 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 IMPLEMENTATION

GRASP is still a very immature editor, and many of its implementation details are likely to change. However, there are certain design decisions that will probably stay fixed throughout the lifetime of GRASP.

2.1 Kawa and JVM inter-operation

GRASP is implemented in – and intimately coupled to – Kawa, the implementation of Scheme which runs on the Java Virtual Machine and produces JVM byte code.

The main reason for this decision is that JVM byte code can be translated to run on Android, which was both the initial development platform of GRASP, as well as its main target.

Kawa Scheme offers a few interesting extensions to facilitate inter-operation with JVM: first, it exposes Java's object model to Scheme (using the `define-simple-class` special form); second, it extends Scheme with an optional syntax for declaring types, and provides a Java-like type system.

GRASP uses Scheme's syntax extension mechanisms to provide two alternative ways of defining new types.

2.1.1 *The record system.*

The first mechanism for defining new types probably also happens to be the first almost decent record system in the history of Lisp.

It lets programmers define record types in the following way:

```
(define-type (Extent width: real := 0
                     height: real := 0))
```

A new instance of a record defined this way can be created by typing, say

```
(define carpet ::Extent (Extent width: 5 height: 10))
```

and the fields can be accessed using Kawa's reader extension:

```
> carpet:width ; => 5
> carpet:height ; => 10
```

GRASP source code also contains a pattern matcher which allows to destructure records defined that way:

```
(define (square-or-rectangle e)
  (match e
    ((Extent width: x height: x)
     `(a square with side length ,x))
    ((Extent width: x height: y)
     `(an ,x by ,y rectangle))
    (_
     'what-are-you-giving-me?)))
```

Records can also implement interfaces and provide methods that can be invoked on them, although the syntax is not entirely satisfactory:

```
(define-type (Move from: Cursor
                   to: Cursor
                   in: pair := (the-document)
                   with-shift: int := 0)
  implementing Edit
  with
  ((apply!)::Cursor
   (let ((item (extract! at: from from: in)))
     (insert! item into: in at: to)
     (cursor-climb-back to in)))

  ((inverse)::Edit
   (match (this)
     ((Move from: `(,s0 . ,source)
            to: `(,d0 ,d1 . ,destination)
            in: document
            with-shift: s)
      (Move from: (recons (+ d1 1) destination)
            to: (recons* s (- s0 1) source)
            in: document
            with-shift: d0)))))
```

2.1.2 Environment-like class definitions. The second syntax provided by GRASP for defining Java classes is the form, which allows for environment-like class definitions:

```
(define-object (ClassName constructor-params ...)::Iface
  (define field ::type initial-value)
  ...
  (define (method args ...)::result-type body ...)
  ...
  (SuperClass constructor-args ...)
  initialization-code ...)
```

This syntax has some quirks, but since the slot and method definitions are syntactically identical to top-level procedure and variable definitions, the refactoring is facilitated, because moving forms between the top level and class definitions requires no additional actions.

It is also noteworthy that both type definition mechanisms deliberately limit the expressiveness of the raw Java-style class definition (for example, by letting only one constructor to be present in the definition).

2.2 The document representation

Documents in GRASP are essentially represented using `cons`-cells. There are some caveats to this, though.

First, GRASP does not use the implementation of `cons`-cells provided by Kawa: instead, it sub-classes Kawa's `gnu.lists.Pair` class, and adds two significant modifications:

- it overrides the `getCar` and `getCdr` methods (which are internally invoked upon calling `car` and `cdr` in Scheme) so that their behavior depends on value of `(the-cell-access-mode)` parameter:
  - if the value of `(the-cell-access-mode)` is `CellAccessMode:Editing` (the default), then instead of returning atomic values such as symbols or numbers, the `car` and `cdr` functions will return proxy values of type `Atom`;
  - otherwise, if the value of `(the-cell-access-mode)` is `CellAccessMode:Evaluating`, then the actual Scheme atoms will be returned by the `car` and `cdr` operations
- it overrides the `equals` method to use the default Java's object identity (rather than Scheme's `equal?`-like identity provided by Kawa)

The reason why the `equal?`-like identity is unsatisfactory is because GRASP uses Java implementation of weak-key hash tables in order to represent white-space and comments between the elements of a `cons`-cell. (This representation was initially chosen because most implementations of Scheme do not allow to modify the representation of `cond`-cells, so hash-tables seemed to be the only viable choice.)

In the parlance of GRASP, weak-key hash tables are called "properties", and GRASP provides a fairly elegant way of expressing them, using the so called "getters with setters".

More specifically, there are five properties to describe how `cons`-cells are formatted:

(1) `post-head-space`, which designates a `Space` following cell's head,
(2) `pre-head-space`, which designates a `Space` between an opening parenthesis and the first element of a list (and as such it doesn't matter for most cells),
(3) `pre-tail-space`, which – in the case of dotted pairs designates a `Space` following the "dot"
(4) `post-tail-space`, which – in the case of dotted pairs designates a `Space` following the pair's tail
(5) `dotted?`, which designates a Boolean value that specifies whether a given cell should be considered as dotted.
    This stems from the fact that from the perspective of a Lisp reader, the notation `(a b c d)` is indistinguishable from `(a . (b . (c . (d . ()))))`.

`Space` objects designated by the first four properties are objects that contain a single list. The list needs to contain at least one integer number, which signifies the number of spaces that ought to be inserted at a given place. The presence of two consequent numbers signifies a line break – so while the list `(1)` means a single horizontal space, the list `(0 0)` means a line break.

In addition to integers, the list can also contain three types of compound objects, which represent three types of comments defined by Scheme – line comments, block comments and expression comments.

A GRASP document is wrapped in an additional `cons`-cell (whose `cdr` is meaningless), so that it is possible to represent comments and white-space of an empty document, as well as express editing operations in a uniform way.

2.3 The cursor representation

The representation of a cursor in a text editor is fairly straightforward: it is sufficient to provide a line number and a column to characterize the location of a cursor in a file.

In the case of a tree editor, a cursor needs to be expressed as a path on a document tree – a list of indices of expression at the subsequent levels on the tree.

The indices can be used to refer to the elements from a list, but they can also be used to refer to white-space between the elements on the list. If we take, say, an improper list `(a b c . d)`, then index 0 refers to an empty space (pre-head-space) between the opening parenthesis and the element `a`, index 1 refers to the element `a`, index 2 – to the space (`post-head-space`) following the element `a`, index 3 – to the element `b`, index 4 to the space following the element `b`, index 5 to the element `c`, index 6 to the space following the element `c`, index 7 – to the head/tail separator (dot) pseudo-element, 8 to the space following the dot (`pre-tail-space`), index 9 – to the element `d` and index 10 – to the space following `d` (`post-tail-space`).

In the above list, two non-numerical indices are legal: the index `#\[` refers to the opening parenthesis, and the index `#\]` – to the closing parenthesis. Those elements cannot be individually picked up, but the keyboard cursor can be positioned on them.

Of course, in order to be able to refer to an element in a nested structure, a sequence of indices is needed.

In GRASP, such a sequence needs to be decoded from right to left – the rightmost index selects the expression from the top-level. (Actually, since the documents in GRASP are wrapped in an additional cons cell, the rightmost index is always 1.)

This strategy allows to maximize structural sharing between cursors and to prevent garbage generation by utilizing hash-consing. (This seems important, because e.g. the rendering function generates cursors to all rendered elements of the tree.) Hash-consing requires that cursors are treated as immutable.

The function for referring to an element by cursor could be defined in the following way:

```
(define (cursor-ref document cursor)
  (match cursor
    ('()
     document)
    (`(,head . ,tail)
     (let ((parent (cursor-ref document tail)))
       (part-at head parent)))))
```

where `part-at` is a polymorphic function that selects a sub-element from a given element. For lists, `part-at` is defined so that it conforms to the indexing scheme mentioned above, where even indices refer to spaces, and odd indices refer to actual elements.

For the `#\[` and `#\]` indices, the function `part-at` returns the list itself. Likewise, for any atom, the function `part-at` will always return the atom itself.

A cursor such that

```
(eq? (cursor-ref document cursor)
     (cursor-ref document (cdr cursor)))
```

is considered to be fully expanded.

In practice, most operations in GRASP require fully expanded cursors, and keyboard navigation procedures should only generate fully expanded cursors.

2.4 The "undo" mechanism

The documents in GRASP are considered mutable, and the editing of a document occurs by means of mutating the 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 operations:

```
(Move from: Cursor
      to: Cursor
      in: pair := (the-document)
      with-shift: int := 0)
;; the Move operation is its own inverse
```

```
(Insert element: (either pair HeadTailSeparator)
        at: Cursor := (the-cursor)
        into: pair := (the-document))

(Remove element: (either pair
                         HeadTailSeparator
                         EmptyListProxy)
        at: Cursor := (the-cursor)
        from: pair := (the-document)
        with-shift: int := 0)
;; the Move and Remove operations are mutually inverse
```

```
(ResizeBox at: Cursor := (the-cursor)
           from: Extent
           to: Extent
           in: pair := (the-document)
           with-anchor: real)
;; the ResizeBox operation is its own inverse
```

```
(InsertCharacter list: (list-of char)
                 after: Cursor := (the-cursor)
                 into: pair := (the-document))

(RemoveCharacter list: (list-of char)
                 before: Cursor := (the-cursor)
                 from: pair := (the-document))
;; InsertCharacter and RemoveCharacter are mutually inverse
```

```
(SplitElement with: Space
              at: Cursor := (the-cursor)
              in: pair := (the-document))

(MergeElements removing: Space
               at: Cursor := (the-cursor)
               in: pair := (the-document))
;; SplitElement and MergeElements are mutually inverse


(NoEdit)
;; the NoEdit operation is its own inverse
```

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 EVIDENT PROGRAMMING

One of the deep philosophical underpinnings of GRASP is the idea that code is easier to study when it's accompanied by concrete, tangible examples.

For example, consider the famous definition:

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

Even though it consists of only three simple terms, comprehending the above code can be very puzzling. However, if we consider that

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

then its purpose becomes clear immediately, even though the name `f` isn't particularly well chosen.

This style of programming – where definitions are interleaved with usage examples – can be practiced in most implementations of Lisp – but only as long as it concerns things that can be expressed as textual objects.

However, once we enter the realm of things which cannot be expressed as textual objects, the examples become incomprehensible.

Consider, for example, the following list of complex numbers:

```
(0.5403023058681398+0.8414709848078965i
-0.4161468365471424+0.9092974268256817i
-0.9899924966004454+0.1411200080598672i
-0.6536436208636119-0.7568024953079282i
 0.28366218546322625-0.9589242746631385i
 0.960170286650366-0.27941549819892586i
 0.7539022543433046+0.6569865987187891i
-0.14550003380861354+0.9893582466233818i)
```

It may not be immediately apparent that all these points lie on a unit circle – which would be obvious if only they were plotted on the screen.

And obviously, there exist specialized tools which allow for that. But they are rarely integrated with code editors in a way that wouldn't harm the workflow of people who do not use those code editors.

The author of GRASP calls the example-rich approach "evident programming", because it is an attempt of making all components of a program as simple and tangible as possible.

As a matter of fact, GRASP itself attempts to exploit this approach to the extent that is possible in the medium of text. In addition to terminal client, desktop client and Android client, GRASP provides a back-end for rendering documents as strings. This allows to write tests in the following form:

```
(insert-character! #\[)

(e.g.
 (snapshot) ===> "
┌  ┐
│  │
└ |┘
")
(undo!)

(e.g.
 (snapshot) ===> "
")
(redo!)

(e.g.
 (snapshot) ===> "
┌  ┐
│  │
└ |┘
")
(for-each insert-character! '(#\d #\e #\f #\n #\e))

(e.g.
 (snapshot) ===> "
┌       ┐
│ defne │
└      ^┘
")
(times 2 move-cursor-left!)

(e.g.
 (snapshot) ===> "
┌       ┐
│ defne │
└    ^  ┘
")
```

The author believes that such tests should be approachable not only to people who are new to the GRASP code base, but also to people who are new to programming. They are a good starting point for understanding the roles of particular operators defined in the GRASP code base.

4 DEVELOPMENT HISTORY

The development of GRASP begun in the late 2018 with the announcement of "The Draggable Rectangle Challenge". The first prototype was created in Racket at the beginning of 2019, and was presented during that year's spring edition of RacketFest in Berlin, although it wasn't an actually usable program.

In the meantime, the author's computer broke down due to some unfortunate accident, and the only programmable device that the author was left with was his Android phone.

Since the author managed to figure out how to build Android applications on Android (using the Termux app), he decided to write a prototype of a touchscreen-based editor.

The first iteration was written in the spring of the pandemic year 2020, entirely on the phone, entirely in the Java programming language (TM). It allowed its users to create boxes and textual symbols, but it didn't allow them to save or load files, scroll documents or evaluate expressions, so it was essentially a toy, and its user interface was very clumsy. Its architecture was also willy-nilly, and it quickly turned out that it cannot be further developed.

So in the beginning of 2021 the author decided to start the project over.

By the end of the summer, he managed to build an editor which could open and save files, split screen, edit multiple files and support a rich set of gestures. This version of the editor was presented during the 2021 Scheme Workshop/ICFP. After the presentation, the author also managed to integrate the editor with Kawa Scheme, and presented it at a local Hackerspace event in Poland, where it was very well received.

However, even though the editor is an actually usable applications, it also turned out to suffer from a number of shortcomings. First, it was only able to represent proper lists and atoms, and adding support for improper lists, strings and comments seemed difficult. Second, it did not support cursor, and supporting it also seemed difficult. Third, it was based on direct manipulation of expressions, and there was no obvious way of adding the "undo" feature. Most importantly, the editor was written in Java, which meant that – being an s-expression editor – it could not be used to further develop itself (and the inclusion of the Kawa compiler significantly increased its build times).

So in the beginning of 2022 the author decided to start the project over.

This time it was envisioned as a cross-platform project from the beginning. However, the only "common platform" between Android and PC known to the author was VT-100 terminal emulator, which determined that – in addition to its rich graphics-based clients, GRASP should also provide a terminal client, which – even if not as capable as its more advanced siblings – would still be useful.

Besides, adding it was fairly simple, given that a lot of functionality is shared with a string-rendering back-end which has been used for testing.

So far, the development of the latest iteration of GRASP already took over a year, so in the beginning of 2023, the author has decided not to start the project over, which he believed was a good sign.

5 CURRENT PROGRESS

Although this paper could leave a different impression, at the moment of writing (February 2023) GRASP isn't yet a usable application – in some ways it is inferior to the Java-based Android client that was demoed in 2021 in that:

- it doesn't let open or save files
- it doesn't let split or scroll the screen
- it doesn't let 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

6 RELATED WORK

The strongest source of inspiration for GRASP has been Emacs, 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.)

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.

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

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

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).

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

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

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

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.

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

https://marketplace.visualstudio.com/items?itemName=hediet.debug-visualize