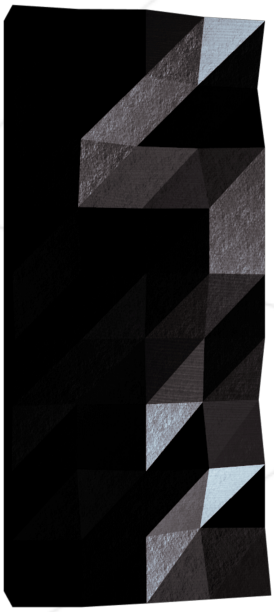


Carnifex Project

A Conway's automaton

Pedago pedago@42.fr

Summary: This time, let's code a Game Of Life, also known as Conway's automaton. And while we're talking about Life, let's do it in Lisp.

Contents

I	**Forewords**	**2**
II	**Introduction**	**3**
III	**Goals**	**5**
IV	**General instructions**	**6**
V	**Mandatory part**	**8**
VI	**Bonus part**	**10**
VII	**Turn-in and peer-evaluation**	**11**

Chapter I

Forewords

Figure I.1: These guys are as awesome as this project.

Chapter II

Introduction

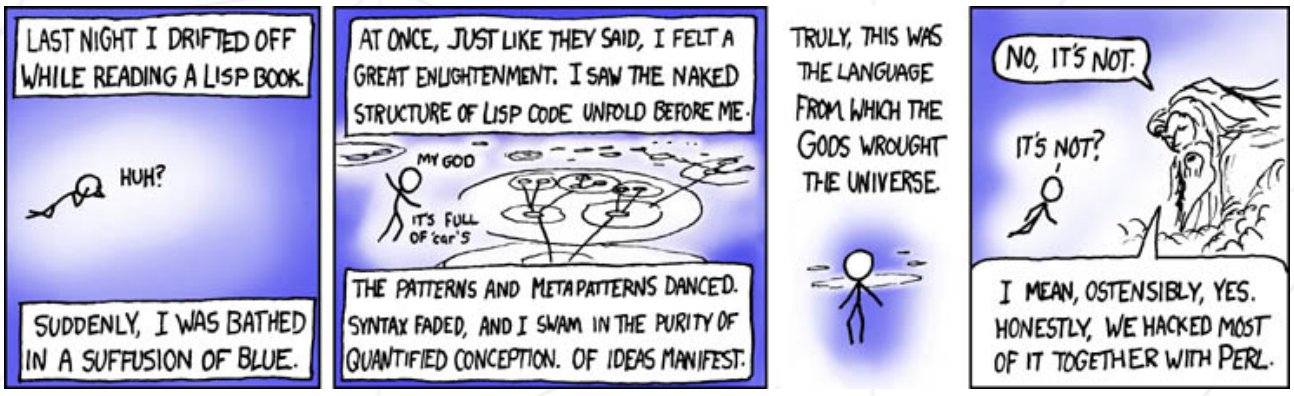
In this rush, you have the chance to (re)discover two amazing elements of computer science : Conway's automaton, and the `Lisp` programming language.

Conway's automaton, or "Game Of Life", is a cellular automaton whose initial input configuration determines its evolution. The game is played on a virtually infinit grid of cells, each cell beeing either alive or dead. The life or death of a cell is determined by the state of its 8 neighbors according to a set of simple rules. Have a look to these links to get a better idea :

- https://www.youtube.com/watch?v=C2vgICfQawE
- http://www.bitstorm.org/gameoflife/

Mesmerizing isn't it ?

Now let's talk about `Lisp`. Why would you like to learn `Lisp` in the first place ? Well, for those among you familiar with Randall Munroe's work, the answer is obvious. For those who don't, here is why :



Now you know why. But there's still work to do : actually learn Lisp. Here is everthing you want to know about Lisp. Wether you decide to read the full book or not, please read at least this and this sections.

Chapter III

Goals

`Lisp` and Conway's automaton are really two fascinating elements of computer science, even if they have nothing in common. Except this projet maybe... As a consequence, there are two goals to this project: discovering `Lisp` and writing your first Conway's automaton.

`Lisp` is an amazing language, really. The swarm of parenthesis might be a little bit difficult to wrap your head around at first, but once you get it, a new world opens. Its functionnal paradigm, the way "everything" is a list, the way the list are processed, handling data as code and vice versa, etc. Beeing able to use `Lisp` and to understand it gives you an edge when adressing any programming language.

When first watching a Conway's automaton unfold, one is mesmerized. Seeing all these little cells live and die, again and again, at high speed while creating complex and seemingly "alive" patterns really reminds of actual living cells. Things really get weird when one discovers that, through this entertaining chaos, order can emerge. Specific patterns can be used to create complex interractions between blobs of cells, self replication, or message passing for instance. Lot's of sorcerer's apprentices try day after day to create more and more complex interactions and use Conway's automaton as a rightful computing model.

Why not you ?

Chapter IV

General instructions

- You must use the `Lisp` interpreter `sbcl`. The version 1.2.11 is tested and validated on the school's dump.
- The graphical display of your automaton must use the `SDL` bind for `Lisp` `"lispbuilder-sdl"`.
- For your convenience here are the setup instrutions. Root privileges are never required.
  - If not already done, install the `SDL` on your session using the MSC.
  - Fetch `sbcl` binary distribution here.
  - Create an environnement variable `INSTALL_ROOT` with value `"$HOME/.sbcl"`.
  - Untar the tarball and execute the `install.sh` script.
  - The `Lisp` interpreter `sbcl` is now installed on your session in `"$HOME/.sbcl"`. As this path is not the default one expected by `sbcl`, create the environnement variable `SBCL_HOME` with value `"$HOME/.sbcl/lib/sbcl"`.
  - Add `"$HOME/.sbcl/bin"` to your `"$PATH"`, alias `"sbcl --noinform"` to `"sbcl"` to get rid of the greeting header, and you're ready to use `sbcl`.
  - Now, let's add `quicklisp`, a package manager, similar to `pip` for `python`, `gem` for `ruby`, or `opam` for `ocaml`. `quicklisp` is available here.
  - From now on, any command starting with the `'*'` character refers to a `Lisp` command in `sbcl`.

```
$> sbcl --load quicklisp.lisp
* (quicklisp-quickstart:install :path "/<path/to/your/home>/.sbcl/quicklisp")
* (ql:add-to-init-file)
```

  - Please make sure that you use the complete path like '/nfs/zfs-stud.....'.
  - Your packages manager is ready to add libraries to your `sbcl`. Let's install the `SDL` bindings for `Lisp`. For a very specific reason, the next command **WILL** fail, but run it anyway, and when it fails, choose the abort option and exit `sbcl`.

```
* (ql:quickload "lispbuilder-sdl")
```

- What happened ? `quicklisp` first downloads the package, then tries to install it. `OSX` beeing the operating system it is, a small part of the `SDL` binding must be compiled by hand before `quicklisp` can install it (on `OSX` and `OSX` only, every other operating system I've used `Lisp` with just handle this fine. The more recent `OSX` distribution you use, the worst it will be). That's why we wanted to run the command one time even if it fails : `quicklisp` fetches the package, you compile the small `OSX` dependant part, then run the installation again via `quicklisp`, succesfully this time.

```
$> cd ~/.sbcl/quicklisp/
$> cd dists/quicklisp/software/lispbuilder-<version>/lispbuilder-sdl/cocoahelper
$> make
$> sbcl
* (ql:quickload "lispbuilder-sdl")
```

If you have an error that you can't explain, or if you are completly lost, just delete your $HOME/.sbcl folder and start all over again.

Chapter V

Mandatory part

Now the cool part:

- Write a Game of Life in `Lisp` using `lispbuilder-sdl` for your display and inputs.
- Many variants of the game exist. You must use the standard set of rules:
  - Any live cell with fewer than two live neighbours dies, as if caused by under-population.
  - Any live cell with two or three live neighbours lives on to the next generation.
  - Any live cell with more than three live neighbours dies, as if by over-population.
  - Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.
- For obvious reasons, the grid can't really be inifinit. Otherwise some moving patterns will skim through your entire memory, and you don't want that. The grid size must be specified as parameters to your program. Any cell beyond your grid is always dead when testing cells at the edges.
- When run without any parameters, your program must display a usage. For instance :

```
$sbcl --load game_of_life.lsp
usage: sbcl --load game_of_life.lsp [-h] width height

positional arguments:
  width                 width of the grid

  height                height of the grid

optional arguments:
  -h, --help            show this help message and exit
```

- It must be possible to close the window and exit the program by clicking the red cross atop the window or by pressing the `'esc'` key.
- It must be possible to skim through your grid using the `w`, `a`, `s` and `d` keys if the grid is larger than the display window. This behaviour must also be available by draging the display with the mouse.

- It must be possible to zoom the display in and out by using the `'+'` and `'-'` keys. This behaviour must also be available by using the mouse wheel.
- Pressing the `'p'` key pauses time. Pressing it again unpauses time. The game starts paused.
- It must be possible to slow and speed up time by using the `'<'` and `'>'` keys. This behaviour must also be available by using the mouse wheel while holding the `'shift'` key.
- It must be possible to set a cell alive or dead by clicking on it. Thus when launching the program with an empty grid, one can create an initial state using the mouse.
- It must be possible to reset the game to an empty grid by pressing the `'r'` key. Reseting the game also pauses it.

On using SDL in Lisp on modern OSX distributions, you may encounter a run time error FLOATING_POINT_INEXACT although your code is correct. To avoid it, wrap the entry point of your code as follows : (sb-int:with-float-traps-masked (:invalid :inexact :overflow) (main *posix-argv*)) instead of just writing (main *posix-argv*) for instance.

Chapter VI

Bonus part

And now, the cooler part. The limit is just you imagination. Here are some ideas :

- Coloring dead cells that were alive at least once to have a visal representation of the "fingerprint" of the intial state after stabilization.
- Toggle the display of the grid layout.
- A small GUI able to add specific patterns at wanted positions, such as `pulsars`, `gliders`, `gliders guns`, `lightweight spaceships`, ...
- Implementing the `hashlife` algorithm.

Chapter VII

Turn-in and peer-evaluation

Turn your work in using your `GiT` repository, as usual. Only work present on your repository will be graded in defense.