

# Conway's Game Of Life

March 21, 2014

Brown, Skylar -- skmbrown@ucsc.edu

Fukano, Mat -- mfukano@ucsc.edu

## Project Overview

Our objective was to implement Conway's Game of Life in three different languages: C, Haskell, and Scala. Computing problems are universal, and by working on them in different languages, we can get better understand the specifics of and unique features of each language. By comparing how each language solves the same problems, we can uncover objective and subjective advantages and disadvantages of each.

## Motivation

Learning a new programming language opens up many possibilities and can expand your way of thinking. There are many specifics that are included in a language either as a feature or a defining point of individuality that can provide insight into new and different ways to use programming.

We wanted to learn new languages that we were unfamiliar with and put our capabilities to the test by implementing something that was familiar to us. As a method of comparison, we used C as a familiar baseline as both members of our team have used it extensively in the past. Haskell was chosen as a middle ground as we have been learning the basics of Haskell throughout the quarter yet are still fairly new to it. Scala was chosen as a completely new language that we would have to learn from the ground up on our own.

## Development

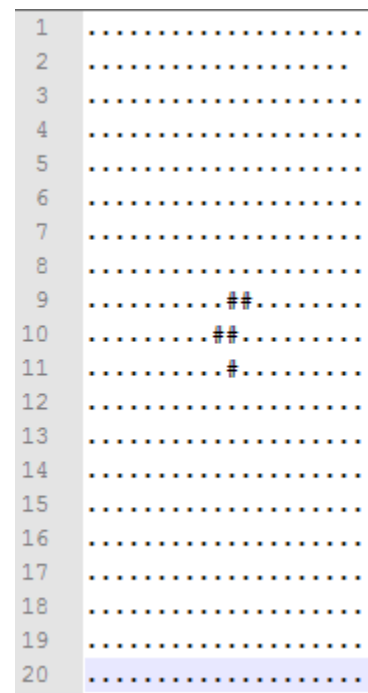
The way we designed the project was helpful while implementing it. We had three languages and knew how to approach the project, moving forward with each part could be done modularly. There were a couple of different stages in design: code structure, file input and board output, game logic, and language specific code segments. We also had stretch goals that we were aiming for, however there ended up being plenty of work and last minute complications with just implementing our initial goals.

## Code Structure

Code structure and length was very different between the three languages. Haskell did not require creating any code structures, just a collections of interconnected functions. C required only a little bit more planning in terms of how to organize the code as we used a smaller number of functions than Haskell and each function had more lines. Scala required a lot of planning ahead and rearranging. We ended up settling on a structure where the world is implemented as an object with private methods and variables within.

## I/O

There are many different ways to print out a game board in Game of Life. What's the best way? Do you want static states so that you can compare where things actually occur? Would it be better to have a GUI that would display things easily and accept input? Is it better to pass an initial state and allow it to develop on its own or to have a random seed? These are all things that we considered. We initially implemented a series of static board states like [1] that would print one at a time based on user confirmation. This idea was clunky, and ended up being too slow if the board design was more complex and interesting. Our final implementation uses an in-console board that runs continuously, using unicode blocks that define the cells in a more uniformly spaced grid. It was a large visual improvement and made the game feel more alive. In terms of I/O in each language, C was by far the hardest. It required over 30 lines of code to simply read and analyze the contents of a text file. Haskell and Scala were both a much more reasonable 3-5 lines as both have built in library functions of file I/O



[1]: Initial state that we pass in

## Game Logic

C was the language that gave us the least difficulty in programming. This is likely due to us being much more familiar with C than any other language. A feature of the language that we found most helpful in comparison to Haskell and Scala was the simplicity and mutable nature of arrays in C. By using a two-dimensional array of cells for the game board, it was just a matter of applying the rules of evolution to each cell in the array and updating its state to be either alive or dead. Iteration over the board was done with two nested for-loops, and the rules enacted on each space were performed in a similar manner.

The Haskell code uses arrays as well, but indexing into them was more roundabout, as data is immutable in Haskell. We couldn't do anything as simple as `propagate(array[index])` in a loop while incrementing the index, like we could in C. Instead, we used a list comprehension

technique that takes in a game-world and creates a new list of alive or dead cells by applying the rules of evolution to each cell [2]. We then had to feed this list into an Array constructor function that would convert the list into a 2 dimensional array. This ended up being fairly comparable to the way we implemented in C with only one extra step of having to use an array constructor instead of simply changing the values in the array. It took much fewer lines of code but ran much slower than C.

```
27 evolveWorld :: World -> World
28 evolveWorld w = listArray (bounds w) newWorld where
29     size = getSize w
30     newWorld = [evolveCell w x y | x <- [1..size], y <- [1..size]]
```

[2]: Propagation rules in Haskell

The Scala implementation was much different from the Haskell and C versions. We used a world class which contains private functions that operate on the contents of the game-world, whereas in the other languages we simply used a collection of functions. To represent the game board, we used a two dimensional array built from a list, similar to Haskell. We scanned the text file for alive cells and pushed them one by one to a list of coordinates. We then used that list of alive coordinates to create a game-world with alive or dead cells that could be evolved. This differed from the other two languages as we did not use an intermediary list of alive coordinates in the other languages.

### *Timeline*

Modular design was always a goal with this project. We began by envisioning where we wanted to be with the project over the course of the quarter. The goal was to first develop a properly working version in C, and then move onto Haskell and see what time permitted. We ended up with fairly functional prototypes at an early enough point, but were overconfident with how much time we had. Subsequently, we shifted our attentions to midterms and other school projects instead of dedicating time towards our stretch goals. Ideally we would have achieved our stretch goal of a Turing complete version of the Game, where the board is dynamically sized by the evolutions of the cells instead of being statically sized and limited by borders of the grid.

## **Challenges**

In comparison to how straightforward C was, Haskell presented some unique and interesting challenges. First off was array accessing without being able to change variables, as discussed in the *Development* section. Secondly, the GHCi shell is a different approach to development than the compiler we are accustomed to in C. This caused some issues with display formatting and presented us with some unexpected problems. As a result, we detracted some time from the Scala development to try and fix visual glitches with the way Haskell was

displaying the continuous run of the Game.

Another interesting issue that came up was that the amount of time we put into working on Scala. The shorter period of development influenced our implementation in the language. The code looked and felt very similar to Java. We didn't branch out in the language as much as we could have, such as taking advantage of its functional properties. We have problems with formatting the output, as we can't figure out how to display unicode characters as easily as the other two languages. Also an interesting speedbump in Scala was casting strings to integers. This required making a custom object [3] and helper function, whereas this functionality was built into Haskell and was fairly easy in C.

```
76
77 object Int {
78     def unapply(s : String) : Option[Int] = try {
79         Some(s.toInt)
80     } catch {
81         case _ : java.lang.NumberFormatException => None
82     }
83 }
84
```

[3] Required helper object in Scala

## Conclusion

The approach for handling a project that implements the same thing in multiple languages was different than we expected. A key point in accomplishing the end goal was to divide the workload into steps and proceed forward with one language at a time. Each of the languages is different, and it can be complex and stressful to have two or three partially-working projects, all with different issues and solutions. However, it was a learning experience, and by being able to choose any language we wanted, it was a much more self-driven learning process. We learned how to problem-solve incrementally and were inspired to learn about the languages in a naturally progressing timeline. The result was a positive experience which taught us how to cope with a large-scale, multi-paradigm programming project, and a sparked curiosity to learn more about both established and developing programming languages.