From Bits to Cells: Simple Cellular Automata in Haskell

Paul R. Potts

December 2006 and March 2025

In 2006, after publishing the blog posts that contained my dot matrix printhead toy, I continued teaching myself Haskell by playing with simple cellular automata, and continued to get helpful feedback and suggestions in the comments. This article contains the original source code, with some minor cleanups to make it work with the current version of GHCi, and with some reformatting as done by the latest Haskell plug-ins for Visual Studio Code when I asked it to "Format Selection." My web template does not play nice with very wide lines of preformatted text, so in some cases I have added line breaks to the output from GHCi; when you execute the statements yourself, it will not have these line breaks.

*In case you're new to Literate Haskell: the Literate Haskell (**.lhs**) file format was inspired by Donald Knuth's Literate Programming concepts and support writing a program in which the comments and code essentially switch places. Instead of writing files of code with a few comments, the developer writes a file containing descriptions of the program and the code, with the actual code embedded in the file like annotations on the text.*

Part One

The little Haskell toy I wrote to simulate a dot-matrix printhead was useful in one sense: it helped me learn a little more Haskell! I couldn't help noticing that the results look like the transformations you get when applying simple cellular automata rules. So my next experiment is to implement those rules, and see what else I can learn from the results.

What follows is Literate Haskell. You can copy and paste the text below into your favorite editor and save it as a file with the extension **.lhs**, then execute it using GHCi. Or, you can access the file directly, using this link. Note that this file does not define a Main, so there's no main function to run. It's made so that you can load some functions and experiment with them interactively by typing statements into GHCi.

From Bits to Cells: some Haskell code to implement
simple cellular automata rules, round one.

Note: this file does not have a Main function and you
cannot load and run it in GHCi as-is; it's designed
so that you can load it into GHCi with :load, then
experiment with by executing statements as described.

The simplest type of one-dimensional CA behaves as
follows:

- a cell may be on or off (one or zero).

- time ticks by in discrete intervals.

- at each tick, the state of each cell is updated
using a function that maps the current state of the
cell to the left, the cell itself, and the cell to
the right to the new cell value.

Because this value-generating function depends on
3 bits of input, a given CA rule actually consists
of 2^3 = 8 possible mappings. The mappings can be
represented by the binary values 0b000 to 0b111
(decimal 0..7). By Wolfram's numbering convention
for these simple CA, we actually count down from
7 to 0. So, let's say we have a rule that maps each
possible set of left, center, and right values:

0b111, 0b110, 0b101, 0b100, 0b011, 0b010, 0b001, 0b000

to the new cell value:

0b0, 0b0, 0b0, 0b1, 0b1, 0b1, 0b1, 0b0

We can treat the resulting number, in this case
0b00011110, as the rule number. This is rule 30 in
Wolfram's schema; there are 2^8, or 256, possible
rules, numbered from 0 to 255.

First, let's make a function that, given a rule
number and three values (left, center, and right),
returns the new cell state. We turn the three values
into an index to look up the appropriate bit in the
rule number.

Note that modern Haskell tooling in 2025 offers a lot
more in the way of useful refactoring suggestions! I'm
going to leave the 2006 code as it was, though, except
for a few changes that seem to be necessary to get it
to parse correctly with the latest GHCi.

```
>import Data.Bits

>genNextBit :: Int -> (Bool, Bool, Bool) -> Bool
>genNextBit rulenum (left, center, right) = rulenum `testBit` idx
>  where
>    idx =
>      (if left then (4 :: Int) else (0 :: Int))
>        .|. (if center then (2 :: Int) else (0 :: Int))
>        .|. (if right then (1 :: Int) else (0 :: Int))
```

Hmmm... it is lacking a certain elegance, and I
don't quite like the way the indentation rules work
in this case, but let's test it with a function that
generates the 8 rule indices in the form of tuples:

```
>genRuleIndices :: [(Bool, Bool, Bool)]
>genRuleIndices =
>  [ ( (val `testBit` 2),
>      (val `testBit` 1),
>      (val `testBit` 0)
>    )
>    | val <- [(7 :: Int), (6 :: Int) .. (0 :: Int)]
>  ]
```

Now if we write:

```
genRuleIndices
```

we get:

```
[(True,True,True),
(True,True,False),
(True,False,True),
(True,False,False),
(False,True,True),
(False,True,False),
(False,False,True),
(False,False,False)]
```

and if we write:

```
map (genNextBit 30) genRuleIndices

this expression curries us a function (mmm... curry!)
which takes a starting state and applies rule 30,
then feeds it the possible starting states. The
result is:

[False,False,False,True,True,True,True,False]

That looks like the bit pattern for rule 30.
Just for fun, let's confirm by making a function
that will translate a list of output bit values back
into a rule number. The signature should look like
this:

sumBitVals :: [Bool] -> Int

And we want the list

>test_bit_vals = [False,False,False,True,True,True,True,False]

to map back to 30. Take a shot at it yourself; you
might find that the result is instructional. I'll wait.

(Musical interlude).

Did you try it? Let's look at some possible
implementation strategies. We could make a list of the
powers of two and then do some list manipulation to
get the powers of two that correspond to our one-bits
summed up:

>pows_2_8_bits = reverse ( take 8 (iterate (2 *) 1) )

pows_2_8_bits

[128,64,32,16,8,4,2,1]

(An aside: since this is generated by a function that
takes no parameters at runtime, it would seem like a
sufficiently smart compiler could generate this list
and stick it in the resulting program so that no
run-time calculation is required to generate it. Does
GHC do this? I don't know.)
```

Anyway, our implementation. We can tuple up our powers
of two with our bit values:

```
tups = zip pows_2_8_bits test_bit_vals

tups

[(128,False),
(64,False),
(32,False),
(16,True),
(8,True),
(4,True),
(2,True),
(1,False)]
```

Then we can turn this back into a list of only the
powers of two that are "on," and sum the results:

```
sum (map (\ tup -> if snd tup then fst tup else 0) tups )

30
```

It seems like this should be simpler still. I looked
in the standard library for a function to filter one
list using a list of boolean values as a comb, but did
not find one. So let's write one:

```
>combListWithList :: [a] -> [Bool] -> [a]
>combListWithList [] [] = []
>combListWithList ls comb =
>  if (head comb)
>    then (head ls) : combListWithList (tail ls) (tail comb)
>    else combListWithList (tail ls) (tail comb)

combListWithList pows_2_8_bits test_bit_vals

[16,8,4,2]
```

That seems good, although it doesn't express the
idea that the lists have to be the same length. It
still seems amazing to me that I can reel off
functions like that in Haskell and have them work
right the first time! Now we can produce our final
function:

```
>sumBitVals :: [Bool] -> Int
>sumBitVals ls =
>  sum $ combListWithList pows_2_8_bits ls

sumBitVals test_bit_vals

30

There is probably an elementary way to implement
the above function using zipWith instead of my combList
but I'll leave that to you; leave a comment if you
figure it out!

As another aside, here is another function I came up
with to generate these bit vals; it doesn't rely on a
specific length.

>foldBitVals :: [Bool] -> Int
>foldBitVals ls =
>  snd
>    ( foldr
>        ( \flag tup ->
>            if flag
>              then ((fst tup) * 2, (snd tup) + (fst tup))
>              else ((fst tup) * 2, snd tup)
>        )
>        (1, 0)
>        ls
>    )

foldBitVals test_bit_vals

30

Take a moment to understand the above function; it is
a little strange. I perform a fold using a tuple. The
first element is the power of two, so it keeps track
of our place in the list, and the second is an
accumulator as we add up our values. This is a trick
for stashing a state that is more complex than a single
value into a fold.
```

Anyway, that was a long diversion, so let's end for now. Next time we'll look at ways to represent our cellular automaton universe as it evolves by applying our rules. As always, your comments are welcome!

Part One Discussion (From Blogger Comments)

Neil Mitchell wrote:

In reference to my comment "An aside: since this is generated by a function that takes no parameters at runtime, it would seem like a sufficiently smart compiler could generate this list and stick it in the resulting program so that no run-time calculation is required to generate it. Does GHC do this? I don't know."

> I strongly suspect it doesn't. If you did, then you might (at runtime) have to do an unbounded amount of computation. In general its not worth doing — either the computation is small (low runtime cost anyway), or its big (high compile time cost, possibly large amount of result).
>
> Another point, use the power of the pattern match!

```
combListWithList (lh:lt) (ch:ct) =
 if ch then lh : combListWithList lt ct else combListWithList lt ct
```

> (It's also faster.)

```
foldBitVals :: [Bool] -> Int
foldBitVals ls = snd (foldr
 (\ flag (t1,t2) -> if flag then (t1 * 2, t2 + t1)
 else (t1 * 2, t2) )
 (1, 0) ls )
```

> Which, of course, you can rewrite as something like:

```
sum [(a,b) <- zip ls bit_values | a]
```

> (where **bit_values** = [1,2,4,...]___ — similar to as defined previously)

The Alternate Moebyus wrote:

> Paul,
>
> I guess the neatest way to do the base-conversion is something on the lines of (coded without ghc, so it might have a bit of handwaving):

```
b2num :: [Bool] -> Int
b2num = foldl (\ x y -> 2 * x + fromEnum y) 0
```

> I use something similar for the Project Euler problems (incidentally, you might want to check them out — solving those problems is a nice way to learn a new language).

An unknown Blogger user wrote:

In reference to my comment "It still seems amazing to me that I can reel off functions like that in Haskell and have them work right the first time!"

I know! *Squee*

*In reference to my comment "There is probably an elementary way to implement the above function using **zipWith** instead of my **combList**, but I'll leave that to you; leave a comment if you figure it out!"*

> Indeed, that's the approach I took. Since **zipWith** is just **zip** then **map**, you practically wrote it yourself! **zip** is even equivalent to **zipWith (,)** (where **(,)** is the pair constructor, read as "pair"). If you're just going to **map** the pairs together, you needn't construct them in the first place. Here's what I came up with:

```
sumBitVals :: [Bool] -> Int
sumBitVals bits = sum (zipWith intify (reverse bits) powersOfTwo)
where powersOfTwo = map (^2) [0..]
intify bit value = if bit then value else 0
```

> I like the way you used **iterate (*2)** to generate the **powersOfTwo**. I'd have never thought of that. Then again, I'm not used to such a bizarre concept as "iterating" in Haskell. :) Recursion For The Win!

I wrote:

> Hi Alternate Moebyus,
>
> That **b2num** function is crazy… I have to step through the expansion steps to see how it works!
>
> On [], **foldl** doesn't ever call our lambda, so we just get the initial value.
>
> On [**False**], **fromEnum** yields **0**, and our initial **x** is zero, so we get **2 * 0 + 0**, or **0**.
>
> On [**True**] **fromEnum** yields 1, so we get **2 * 0 + 1**, or **1**.
>
> On [**False, True**] we are folding left, so we get **0** from the first pass and then **2 * 0 + 1**, or **1**.
>
> On [**True, False**] we are folding left so we get **1** for the first pass, then **2 * 1 + 0**, or **2**.
>
> On [**True, True**] we get **1** from the first pass, then **2 * 1 + 1**, or **3**.
>
> At first glance this function looks to me like it should not work for all lists… but it does seem to work!

The Alternate Moebyus replied:

> While I'd rather not derive it again :) it's supposed to be the optimized version of evaluating a polynomial (a finite power series, I guess) at 2.
>
> *a0 + 2(a1 + 2(a2 + 2…))*

Part Two

Let's get back to our CA generator. Literate Haskell follows. You can copy and paste the text below into your favorite editor and save it as a file with the extension **.lhs**, then execute it using GHCi. Or, you can access the file directly, using this link.

```
Last time we defined a function to generate the
next state of a given cell in our cellular universe,
given a rule number and a tuple consisting of the
current state of the cell to the left, the cell
itself, and the cell to the right.

>import Data.Bits

>genNextBit :: Int -> (Bool, Bool, Bool) -> Bool
>genNextBit rulenum (left, center, right) = rulenum `testBit` idx
>  where
>    idx =
>      (if left then (4 :: Int) else (0 :: Int))
>        .|. (if center then (2 :: Int) else (0 :: Int))
>        .|. (if right then (1 :: Int) else (0 :: Int))

Recall that we can use automatic currying to make a
rule-applying function like so:

>rule_30 = genNextBit 30

We can ask GHCi for the type:

:type rule_30
rule_30 :: (Bool, Bool, Bool) -> Bool

I've put it off while I work on the rules, but it is
time to figure out how to actually represent our CA
universe. Let's start by using a list. I know that
I'm going to write a number of inefficient functions,
and do evil things like take the length of lists a
lot, but let's suspend all concerns about efficiency
```

over to a future discussion and consider this purely
a proof-of-concept.

Our inital universe at time zero has one cell set
to True:

```
>initial_universe = [True]
```

But that isn't quite the right representation for
the universe, because it implies that our whole
universe is one cell in size. We can't even apply
our rule once because there is no left cell and
right cell! Really, we want to pretend that we
have an _infinite_ universe; at time zero, all the
cells to the left and right hold False. Remember,
Haskell is so powerful that it can traverse an
infinite list in only 0.0003 seconds! Well, if you
don't evaluate the whole thing, that is. Taking
advantage of lazy evaluation, you can define all
kinds of infinite structures. This construct will
give us an infinite list of False values:

```
>allFalse :: [Bool]
>allFalse = False : allFalse
```

We don't want to evaluate allFalse, but we can
partially evaluate it using a function like take.
So can we represent our universe like this?

```
>genInfiniteUniverse :: [Bool] -> [Bool]
>genInfiniteUniverse known_universe =
>  allFalse ++ known_universe ++ allFalse
```

Let's try it:

```
take 10 ( genInfiniteUniverse initial_universe )
```

```
[False,False,False,False,False,
False,False,False,False,False]
```

Nope! Since the left-hand side of the universe is
infinite, we will never reach the middle element,
no matter how far we travel from the start of the
list!

That's no good. However, we can do it another way.

We'll allow our universe to be expanded on demand
on the left and right sides:

```
>expandUniverse :: Int -> [Bool] -> [Bool]
>expandUniverse expand_by known_universe =
>  (take expand_by allFalse)
>    ++ known_universe
>    ++ (take expand_by allFalse)
```

```
expandUniverse 3 initial_universe
```

```
[False,False,False,True,False,False,False]
```

We can use the expandUniverse function to expand
our initial universe out to a standardized width
before we start applying the rules.

First, here's a routine to stringify a universe for display:

```
>boolToChar :: Bool -> Char
>boolToChar True = '#'
>boolToChar False = ' '

>stringifyUniverse :: [Bool] -> String
>stringifyUniverse ls = map boolToChar ls
```

Now our infrastructure is in place, so let's figure
out how to apply our generator rule. We know that we
want to start with our initial universe. Let's expand
it to a fixed size. This will give us enough elements
to start making left/center/right tuples out of each
consecutive set of three cells. Each tuple is then
used to look up the next state of the cell at the
center; this will become an element in our next
universe. Then we move to the next cell (not three
cells down). This means that the tuples overlap.

Let's make the tuples. We have to do some thinking
here and consider all the cases; the behavior isn't
immediately obvious. The following almost works:

```
>universeToTuples1 :: [Bool] -> [(Bool, Bool, Bool)]
>universeToTuples1 universe
>  | length universe >= 3 =
>      (universe !! 0, universe !! 1, universe !! 2)
>        : universeToTuples1 (tail universe)
```

```
>universeToTuples1 universe = []

universeToTuples1 [False, True, True, True, False]

[(False,True,True),
(True,True,True),
(True,True,False)]
```

But it isn't quite right; it leaves off the end cases; when we apply our rules, the intermediate representation of the universe as a list of tuples to look up cell mappings will shrink. We actually want the following tuples:

```
[(False,False,True),
(False,True,True),
(True,True,True),
(True,True,False),
(True,False,False)]
```

where the first element of the list is considered as if it was just to the right of an implied False, and the last element is considered as if it was just to the left of another implied False. This sounds like another place we can use our universe expander:

```
>universeToTuples2 :: [Bool] -> [(Bool, Bool, Bool)]
>universeToTuples2 [] = []
>universeToTuples2 universe =
>  tupleize $ expandUniverse 1 universe
>  where
>    tupleize xs =
>      if length xs > 3
>        then
>          tuple3 xs : tupleize (tail xs)
>        else [tuple3 xs]
>    tuple3 xs = (xs !! 0, xs !! 1, xs !! 2)
```

Why did I write it that way? Well, I tried to write tupleize using guards, special-casing length xs > 3 followed by an unguarded case for all other possibilities, but GHC didn't like it --- it told me I had non-exhaustive patterns. There is probably a smarter way to write this, but note that we definitely don't want this version:

```
>universeToTuples3 universe =
>  (xs !! 0, xs !! 1, xs !! 2)
>    : universeToTuples3 (tail xs)
>  where
>    xs = expandUniverse 1 universe
```

In that version, the universe keeps expanding as
we walk down the list, and we never get to the end!

OK, now that we have our tuples, we want to turn
them into our new universe, given a cellular rule
number:

```
>tuplesToUniverse :: Int -> [(Bool, Bool, Bool)] -> [Bool]
>tuplesToUniverse rule [] = []
>tuplesToUniverse rule (tup : tups) =
>  genNextBit rule tup : tuplesToUniverse rule tups
```

Note that we don't have to explicitly take the
tail since we provide a name for it in the pattern.
We're ready to define our genUniverses function
that applies a given CA rule. We can express a
given generation like this:

```
>nextUniverse :: Int -> [Bool] -> [Bool]
>nextUniverse rule universe =
>   tuplesToUniverse rule $ universeToTuples2 universe
```

Now, let's generalize it:

```
>genUniverses :: Int -> Int -> Int -> [[Bool]]
>genUniverses rule width count =
>  take
>    count
>    ( iterate
>        (nextUniverse rule)
>        ( expandUniverse
>            (width `div` 2)
>            initial_universe
>        )
>    )
```

(You could also use a fold, and I'm sure there are
lots of other ways to do it, but iterate seems to
work fine).

And now, witness the unfolding of a universe! Note
that the parameters that go to genUniverses are the
rule number, the width for display, and the number
of generations:

```
putStr $ unlines $ map stringifyUniverse $ genUniverses 222 19 10
```

I get this back from GHCi. Note that GHCI doesn't
like loading a Literate Haskell file with a line that
starts with a hash mark, so I've added extra spaces
on the left side that aren't there in the output.

```
          #
         ###
        #####
       #######
      #########
     ###########
    #############
   ###############
  #################
 ###################
```

Notice that this looks like the image for rule
222 on:

https://mathworld.wolfram.com/ElementaryCellularAutomaton.html

Yay!

In general, a width of twice the number of
generations - 1 will show all the transitions
we are interested in; you could consider the
diagonal area above to be the "light cone" of
events causally connected to that single point
(although some rules will generate True cells
outside of that "light cone" based on the other
False values in the initial universe). Let's
make a helper function to choose a width for us:

```
>showRule rule gens =
>  putStr $
>    unlines $
>      map stringifyUniverse $
>        genUniverses rule (gens * 2 - 1) gens
```

Let's try a few of the other rules:

```
showRule 252 15
                #
                ##
                ###
                ####
                #####
                ######
                #######
                ########
                #########
                ##########
                ###########
                ############
                #############
                ##############
                ###############
```

```
showRule 78 15
               #
              ##
             ###
            ## #
           ### #
          ## # #
         ### # #
        ## # # #
       ### # # #
      ## # # # #
     ### # # # #
    ## # # # # #
   ### # # # # #
  ## # # # # # #
 ### # # # # # #
```

```
showRule 94 15
               #
              ###
             ## ##
            ### ###
           ## # # ##
          ### # # ###
         ## # # # # ##
        ### # # # # ###
```

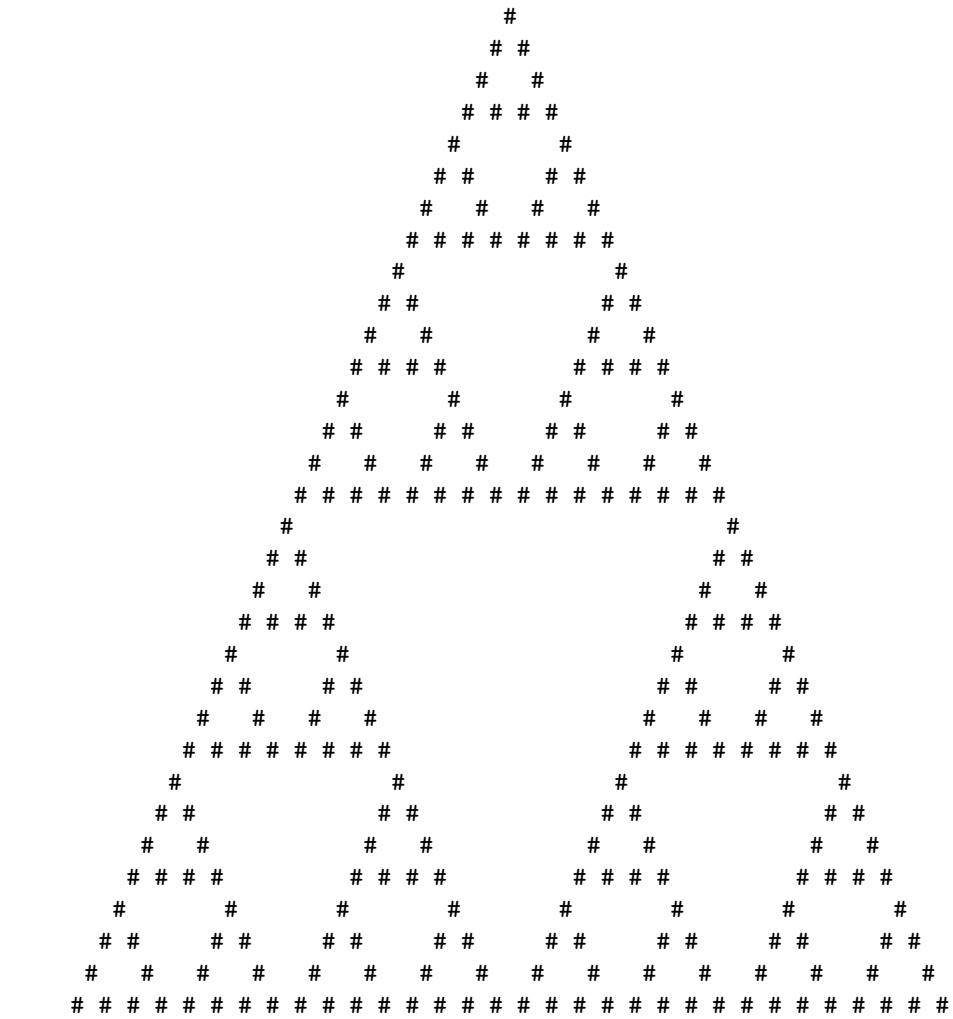
```
       ## # # # # # # ##
      ### # # # # # # ###
     ## # # # # # # # # ##
    ### # # # # # # # # ###
   ## # # # # # # # # # # ##
  ### # # # # # # # # # # ###
 ## # # # # # # # # # # # # ##

showRule 58 15
              #
             # #
            # # #
           # # # #
          # # # # #
         # # # # # #
        # # # # # # #
       # # # # # # # #
      # # # # # # # # #
     # # # # # # # # # #
    # # # # # # # # # # #
   # # # # # # # # # # # #
  # # # # # # # # # # # # #
 # # # # # # # # # # # # # #

showRule 14 15
              #
             ##
            ##
           ##
          ##
         ##
        ##
       ##
      ##
     ##
    ##
   ##
  ##
 ##
```

And finally, my all-time favorite, which generates a low-resolution image of a Sierpinski triangle fractal:

```
>showRule 82 32
```



Wow!

It's not working perfectly, though. Some output doesn't
look right. For example, the output for rule 29:

```
showRule 29 11
          #
 ######### ###########
 #         #
 ######### ###########
 #         #
 ######### ###########
 #         #
```

```
 ######### ###########
 #         #
 ######### ###########
 #         #
```

look different than the way Wolfram's book and web site shows it, which is like this:

```
           #
 ######### ###########
           #
 ######### ###########
           #
 ######### ###########
           #
 ######### ###########
           #
 ######### ###########
           #
```

So there's more to ponder.

Part Two Discussion (From Blogger Comments)

The Alternate Moebyus wrote:

> Hi Paul,
>
> I guess the bug is the truncation of the list at the boundaries? Rule 29 seems to behave something like

```
r29 (False,True,_) = True
r29 (_,_,True) = False
r29 (_,False,_) = True
```

> so with the extended universe having a **[...,False,True,True,...]** at the boundary, it becomes **True**. Not sure if this makes too much sense, though I get (what seems to be) the proper answer by expanding one step more, and then taking the tail of the list before stringify.
>
> Wonderful stuff, though: I've been learning Haskell off and on for quite some time now, but never really had this kind of motivation by example through examples :)

I wrote:

> I experimented with adding in extra **False** values to the left and

> right, but I was not able to get my code to generate results that look like Wolfram's. Now I know why.
>
> It appears that Wolfram's implementation actually wraps at the boundary. According to Wolfram's book A New Kind of Science "we effectively use a cyclic array, in which the left neighbor of the leftmost cell is taken to be rightmost cell, and vice versa."
>
> I'll take a shot at implementing that behavior; I'm not sure I would actually call that solution more correct than pretending we have infinite space available, but it does seem to be more canonical.

Followup to Part Two Discussion

I added these notes to the part two blog post after engaging in some of the discussion in the comments above.

In the discussion above, I mentioned that my code had a bug.

```
showRule 29 11
          #
######### ###########
#         #
######### ###########
#         #
######### ###########
#         #
######### ###########
#         #
######### ###########
#         #
```

look different than the way Wolfram's book and web site shows them, which is like this:

```
          #
######### ###########
          #
######### ###########
          #
######### ###########
          #
######### ###########
          #
######### ###########
          #
```

You can compare the results from my program to the pictures at Wolfram's

MathWorld site here.

After a little investigation, I found that this difference is because Wolfram's implementation *wraps around*; the left neighbor of the leftmost cell of a given universe is taken from the rightmost cell, and vice-versa, while my implementation pretends that there is always more empty space available to the left and right.

The wraparound behavior is probably considered more "canonical," so I came up with a change to make my implementation wrap around. If you replace my **universeToTuples2** function with this one:

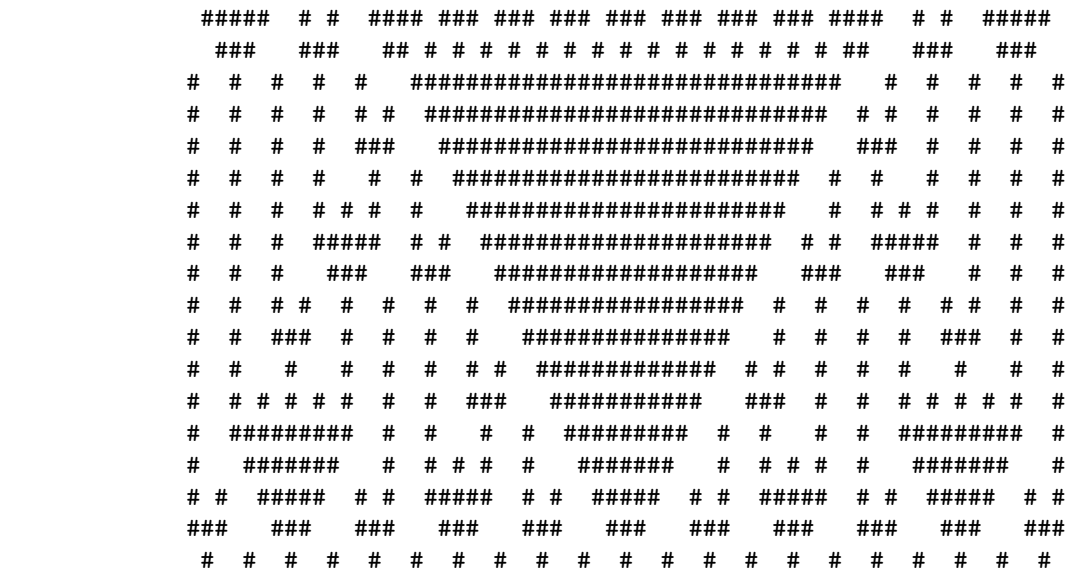
```
universeToTuples2 :: [Bool] -> [(Bool, Bool, Bool)]
universeToTuples2 [] = []
universeToTuples2 universe =
  tupleize $
    wrapUniverse universe
  where
    wrapUniverse xs = (last xs) : (xs ++ [head xs])
    tupleize xs =
      if length xs > 3
        then
          tuple3 xs
            : tupleize (tail xs)
        else [tuple3 xs]
    tuple3 xs = (xs !! 0, xs !! 1, xs !! 2)
```

you will get the wraparound behavior.

I have placed this version of the code in a separate Literate Haskell file here. The only function definition that differs from the version 2 function is the **universeToTuples2** function defined above.

The original version of the code renders rule 165 like this:

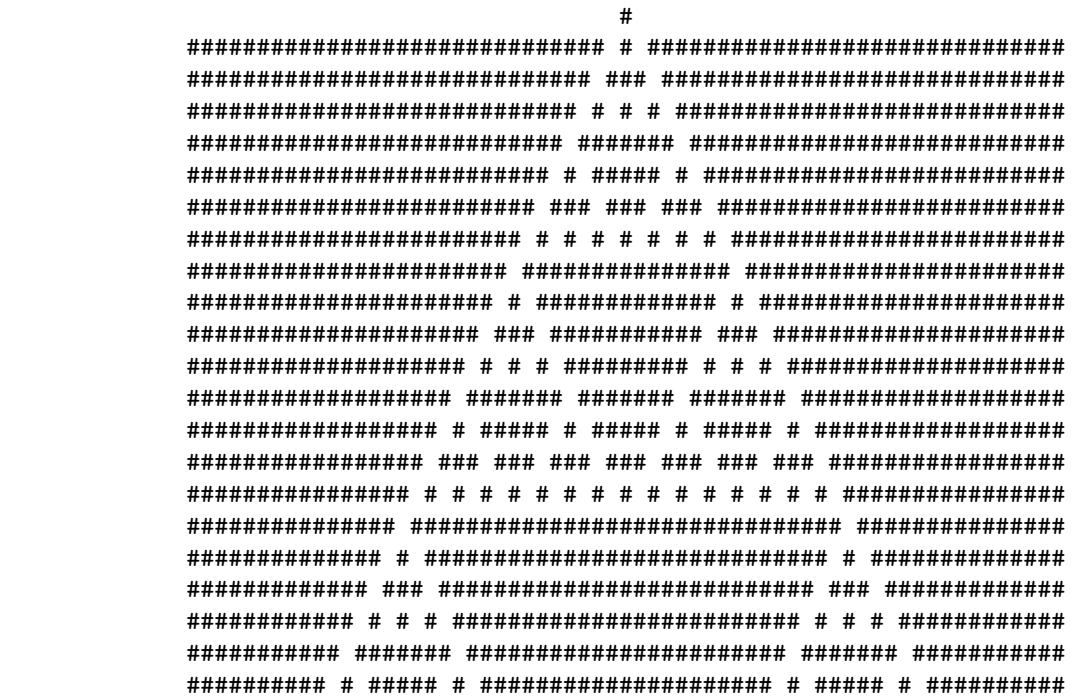
```
showRule 165 32
                                #
 ############################## # ##############################
  ############################ ### ############################
   ########################## # # # ##########################
 #  ######################## ####### ########################  #
 #   ###################### # ##### # ######################   #
 # #  #################### ### ### ### ####################  # #
 ###   ################## # # # # # # # ##################   ###
  #  #  ################ ############### ################  #  #
  #  #   ############## # ############# # ##############   #  #
  #  # #  ############ ### ########### ### ############  # #  #
  #  ###   ########## # # # ######### # # # ##########   ###  #
  #   #  #  ######## ####### ####### ####### ########  #  #   #
  # # #  #   ###### # ##### # ##### # ##### # ######   #  # # #
```



which is interesting in its own right, but doesn't match Wolfram's rendering.

With the revised code, it looks like I expected. The result is (nearly) a photographic negative of the rule 82 Sierpiński triangle:

```
showRule 165 32
```



```
######### ### ### ### ################### ### ### ### #########
######## # # # # # # # ################## # # # # # # # ########
####### ############### ############### ############### #######
###### # ############# # ############# # ############# # ######
##### ### ########### ### ########### ### ########### ### #####
#### # # # ######### # # # ######### # # # ######### # # # ####
### ####### ####### ####### ####### ####### ####### ####### ###
## # ##### # ##### # ##### # ##### # ##### # ##### # ##### # ##
# ### ### ### ### ### ### ### ### ### ### ### ### ### ### ### #
 # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
```

Much nicer!

Thanks for reading! And as always, I appreciate your comments.

Later Comments

Nicholas wrote (in 2022):

> Now that your program is getting more sophisticated, it could benefit from rolling your own types. After all, **stringifyUniverse** shouldn't accept any old list of **Bool**, it should accept a **Universe**!

```
type Cell = Bool
type Universe = [Cell]
type Neighborhood = (Cell, Cell, Cell)
type Rule = Neighborhood -> Cell
```

> Give some helpers to abstract away the Boolness of your Cells:

```
alive = True :: Cell
dead = False :: Cell
```

> Then type your functions accordingly:

```
rule :: Int -> Rule
applyRule :: Rule -> Neighborhood -> Cell
allDead :: Universe
neighborhoodAt :: Universe -> Int -> Neighborhood
universeToNeighborhoods :: Universe -> [Neighborhood]
neighborhoodsToUniverse :: Rule -> [Neighborhood] -> Universe
nextUniverse :: Rule -> Universe -> Universe
```

> That way, your program should be more resilient to underlying type changes. You could change **Universe** to an **Array**, or **Cell** to:

```
data Cell = Dead | Alive deriving (Eq)
```

> And the compiler will tell you what you need to change (which should only be the definitions of alive and dead)! Not to mention that the code reads like natural language!

> The bug in your code is not obvious to me, so I would probably rewrite universeToTuples in terms of this **neighborhoodAt**:

```
u `neighborhoodAt` i = (u' !! i, u' !! i + 1, u' !! i + 2)
where u' = [dead] ++ u ++ [dead]
```

One more thing, if you find yourself writing **(a -> String)** functions a lot, consider making an instance of **Show**:

```
instance Show Cell where
show cell = if cell == alive then "#" else " "

instance Show Universe where
show = concat . map show
```

> Then printing a Universe at the top level will automatically stringify it.

I replied:

> Nicholas, thanks for the wonderful suggestions! That is a lot of new techniques; they will take me a while to digest.
>
> I have added an alternate version of **universeToTuples** that makes my CA generation behave like Mathematica's.
>
> I think I am going to take a break from this for at least 48 hours so I can get some sleep! I want to think over just what I'm going to try next.

Followup in 2025

I now know that this kind of rectangular grid that "wraps around" when moving reaching the North, South, East, or West edge can be implemented by (conceptually) wrapping a 2-dimensional grid around a torus, as described in Richard Bird's paper "On building cyclic and shared structures in Haskell," available here. I have known since childhood that the famous old three utilities problem is impossible on a flat grid, but easy to solve by projecting the grid onto the surface of a torus. So could I just use Bird's data structure? Well, the paper only describes constructing the data structure. I have not looked into what it would take to create updated versions of the immutable toroidal structure on each evolutionary step. It certainly could be an interesting area for futher research. But will it lead to a clear and admirably simple implementation? My magic eight ball says "answer unclear, ask again later!"
