The Dot Matrix Printhead: A Toy to Learn a Little Haskell

Paul R. Potts

December 2006 and March 2025

*In 2006 I published a couple of blog posts where the body of the post was mostly the contents of a "Literate Haskell" file. 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.*

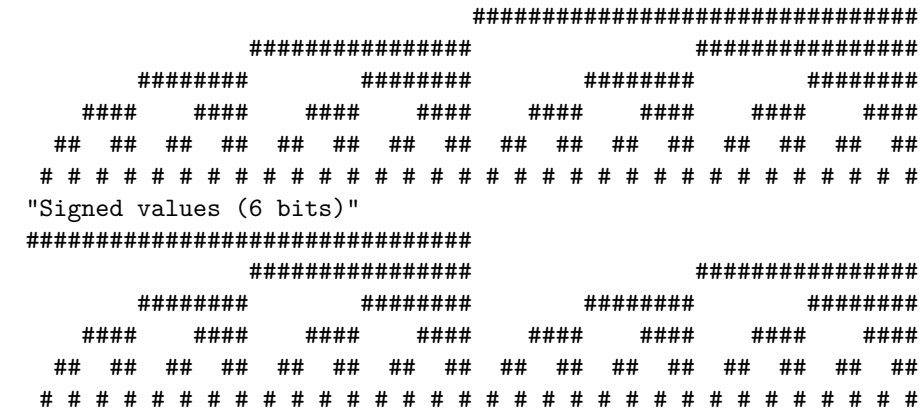
This is a Haskell toy to visualize twos-complement binary numbers. Almost 30 years ago I wrote code on my Radio Shack TRS-80 to make my dot-matrix printer spit out bitmaps of binary numbers, which made pretty recursive patterns. This program commemorates that long-ago geekery from my early teenage hacking days.

I would like to acknowledge the assistance of several people who left helpful comments on the original blog posts. Jason Creighton in particular took a shot at rewriting the code, which was very helpful, and Neil Mitchell provided extensive suggestions as well.

Installing and Running Haskell in 2025

In 2025, I recommend installing a Haskell toolchain using GHCup. Windows users will need the Windows Subsystem for Linux. After installation you should be abel to run **ghci** (this will give you the Glasgow Haskell Compiler as an interactive shell). When I ran this on my little headless Intel NUC running Ubuntu Server, my terminal session looked as follows:

```
paul@balthasar:~/site_src_root/paul/portfolio/haskell$ ghci
GHCi, version 9.10.1: https://www.haskell.org/ghc/  :? for help
ghci> :load dot_matrix_printhead.lhs
[1 of 2] Compiling Main             ( dot_matrix_printhead.lhs, interpreted )
Ok, one module loaded.
ghci> main
"Unsigned values (6 bits)"
```



Version 1

You can find version 1 of the code here, in the form of a **.lhs** file. This is the code as I published it in the first blog post. It is also included below.

```
This is a Haskell toy to visualize twos-complement binary numbers.

In my previous articles on the subject of integer division, I alluded
to underflow and overflow and the "weird number" -- well, now you
can see them!

Almost 30 years ago I wrote code on my Radio Shack TRS-80 to make my
dot-matrix printer spit out bitmaps of binary numbers, which made
pretty recursive patterns. This program commemorates that long-ago
geekery from my early teenage hacking days.

First, we need the bits library:

>import Data.Bits

We'll also need the list library later:

>import Data.List

Let's simulate n-bit binary number in both a signed and unsigned
interpretation. Given a number of bits, we want a function to make
a list of the possible values. For example, in unsigned form 8 bits
will give us a list of the values 0..255; in signed form, it will be
an asymmetrical list from -128..127. Note that this function generates
the whole list, so if you give it a numbits value higher than, say,
16, you might regret it.
```

Our calculation fails for values of numbits <= 0, so we put in a guard case for that. We also want to discourage creating very large lists, so we put in a guard case for numbits > 16.

```
>gen_n_bit_nums_unsigned numbits | numbits <= 0 = []
>gen_n_bit_nums_unsigned numbits | numbits > 16 = error "too many bits!"
>gen_n_bit_nums_unsigned numbits =
>   [ n | n <- [(0::Int)..((2::Int)^numbits - 1)] ]

>gen_n_bit_nums_signed numbits | numbits <= 0 = []
>gen_n_bit_nums_signed numbits | numbits > 16 = error "too many bits!"
>gen_n_bit_nums_signed numbits =
>   [ n | n <- [-(2::Int)^(numbits - 1)..((2::Int)^(numbits - 1) - 1)] ]
```

To test this we can execute:

```
gen_n_bit_nums_unsigned 4
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]

gen_n_bit_nums_signed 4
[-8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7]
```

That looks about right.

Now we want a function to access the bits of those numbers from most significant to least significant and build a list of corresponding boolean values. This would be expressed more naturally if EnumFromTo operated descending sequences. I'm sure there's a better way, but for now I will just reverse the resulting list:

```
>rightmost_bits num_bits val | num_bits > 16 = error "too many bits"
>rightmost_bits num_bits val | num_bits <= 0 = error "too few bits"
>rightmost_bits num_bits val =
>   reverse [ testBit (val::Int) bit_idx | bit_idx <- [0..(num_bits - 1)] ]
```

Once we have a function which will do it with one number, doing it with a whole list of numbers is very easy. We will make a curried function and use map. Currying in Haskell is very easy, which makes sense given that the operation, "to curry" and the language, Haskell, are both named after Haskell Curry!

Our rightmost_bits function takes two arguments, but we want to make a new function which we can hand off as an argument to map, along with a list. This new function will only take one parameter. We can ask GHCi what it thinks the type of the rightmost_bits function is:

```
:type rightmost_bits
```

It says:

```
rightmost_bits :: Int -> Int -> [Bool]
```

We can make a new function out of it by doing partial application:
instead of supplying all its arguments, we just supply the first one.

```
gen_rightmost_bits 4
```

If you do this in GHCi, you'll get an error message because the
result is not printable (GHCi can't apply show to the result).
But you can bind it:

```
let bit_generator_4 = gen_rightmost_bits 4
```

And then ask GHCi what its type is:

```
:type bit_generator_4
bit_generator_4 :: [Int] -> [[Bool]]
```

Take a moment to make sure you understand what we've done here:
Haskell has made us a curried function which takes a list of values
instead of a single value, and generates a list of lists. This
seems pretty close to magic! Let's make another function that
takes the number of bits and a list, generates the curried
function, then maps the list to the curried function:

```
>gen_rightmost_bits num_bits ls =
>   map (rightmost_bits num_bits) ls
```

As you can see, working with curried functions is completely
effortless in Haskell. Here's a general function to generate
a list of all the bit values for unsigned numbers of num_bits
size:

```
>gen_all_bits_unsigned num_bits =
>   gen_rightmost_bits num_bits (gen_n_bit_nums_unsigned num_bits)

>gen_all_bits_signed num_bits =
>   gen_rightmost_bits num_bits (gen_n_bit_nums_signed num_bits)
```

If we execute:

```
gen_all_bits_unsigned 4
```

We get back the following (I have reformatted the output for clarity):

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

This is recognizably a table of the unsigned binary values 0..15. This approach doesn't seem very "Haskell-ish" yet -- instead of generating the combinations as needed, I'm producing them all ahead of time. It also feels to me like I'm pushing uphill on the syntax, and having to use parentheses to force the order of evaluation. We can think more about that later, but let's get our pretty-printing working first.

How can I turn all those boolean values into a string I can print? Well, we can take advantage of the fact that in Haskell, strings can be treated as lists, and the same methods apply. This is not efficient, but it allows us to use some of the standard functional techniques. One of these is "fold."

Fold is a technique for boiling down a list into some kind of summary form. Just what form this summary takes is up to you, since you can supply your own function and a starting value. For example, to sum the elements in a list, I could use foldl like this:

```
foldl (\ x y -> x + y) 0 [0, 1, 2]
```

The first parameter to foldl is a function. In this case I'll use a lambda expression, which gives me a function I won't bother to name. The second paramter is the starting value. The starting value and the next item in the list are repeatedly passed to the function,

which accumulates a value. In this case, the final result is 3.

I can give foldl an empty string as the starting accumulator value and my function can concatenate strings onto it:

```
>stringify_vals =
>   foldl (\ x y -> if y then x ++ "#" else x ++ " ") ""
```

Now you might want to try to apply our list of lists to this function:

```
stringify_vals (gen_all_bits_unsigned num_bits)
```

If you do that, though, you'll get a type error like this:

```
"Couldn't match expected type `Bool' against inferred type `[Bool]'"
```

The reason can be seen if we examine the types:

```
:type (gen_all_bits_unsigned 4) :: [[Bool]]
(gen_all_bits_unsigned 4) :: [[Bool]]

:type stringify_vals
stringify_vals :: [Bool] -> [Char]
```

We have a list of lists of bools, but we want to feed our stringification function a list of bools. To apply each element of (gen_all_bits_unsigned 4) to our stringification function, we can use map again:

```
>stringify_all_unsigned_vals num_bits =
>   map stringify_vals (gen_all_bits_unsigned num_bits)

>stringify_all_signed_vals num_bits =
>   map stringify_vals (gen_all_bits_signed num_bits)
```

This will give us a list of strings, one for each list of boolean values. (I've reformatted the output).

```
["    ",
"   #",
"  # ",
"  ##",
" #  ",
" # #",
" ## ",
```

```
" ###",
"#   ",
"#  #",
"# # ",
"# ##",
"##  ",
"## #",
"### ",
"####"]
```

That's nice: we can start to see our recursive structure! But we really want that value to be turned into a single printable string. To achieve that, we can use the unlines function.

```
>prettyprint_unsigned num_bits =
>   unlines (stringify_all_unsigned_vals num_bits)

>prettyprint_signed num_bits =
>   unlines (stringify_all_signed_vals num_bits)
```

When we interpret

```
prettyprint_unsigned 3
```

we see a single string with newline characters escaped, like so:

```
"   \n  #\n # \n ##\n#  \n# #\n## \n###\n"
```

This will give the results we want when we pass it to putStr, like so:

```
putStr (prettyprint_unsigned 8)
```

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

```
 ####

(etc.)
```

Hmmm. That's nice, but it would be really cool if the results were rotated, so that we could see what it would look like if a printhead moving horizontally was printing out these values. To do this, we can actually start with our intermediate list of lists. If our four-bit numbers give us

```
[[False, False, False, False],
[False, False, False, True],
[False, False, True, False],
etc.
```

we can see the values the other way using transpose. Let's write rotated versions of gen_all_bits_unsigned and gen_all_bits_signed:

```
>gen_all_bits_unsigned_rot num_bits =
>   transpose
>   (gen_rightmost_bits num_bits (gen_n_bit_nums_unsigned num_bits))

>gen_all_bits_signed_rot num_bits =
>   transpose
>   (gen_rightmost_bits num_bits (gen_n_bit_nums_signed num_bits))
```

And make a new stringify function to play with:

```
>stringify_all_unsigned_vals_rot num_bits =
>   map stringify_vals (gen_all_bits_unsigned_rot num_bits)

>prettyprint_unsigned_rot num_bits =
>   unlines (stringify_all_unsigned_vals_rot num_bits)
```

Now

```
putStr (prettyprint_unsigned_rot 6)
```

will give us:

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

Now that's what I'm talking about!

We can use the unsigned version to see the interesting transition point between negative and positive values:

```
>stringify_all_signed_vals_rot num_bits =
>   map stringify_vals (gen_all_bits_signed_rot num_bits)

>prettyprint_signed_rot num_bits =
>   unlines (stringify_all_signed_vals_rot num_bits)
```

Now

```
putStr (prettyprint_signed_rot 6)
```

will give us:

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

Now let's give our program a main function using the IO monad and have it print out several of our results:

```
>main :: IO ()
>main = do
>   print "Unsigned values (8 bits)"
>   putStr (prettyprint_unsigned 8)
>   print "Signed values (8 bits)"
>   putStr (prettyprint_signed 8)
>   print "Unsigned values (6 bits) rotated"
>   putStr (prettyprint_unsigned_rot 6)
>   print "Signed values (6 bits) rotated"
>   putStr (prettyprint_signed_rot 6)
```

After using GHCi's :cd and :load command to read this file, just type:

```
main
```

Discussion of the First Version (From Blogger Comments)

You can find the first version of the program here.

Blogger user Jason rewrote the program and included a link to a blog post containing his own version. I wound up incorporating his changes in version 2. His old link is dead now.

Blogger user Imam Tashdid ul Alam suggested using short function names, and also recommended using

```
foldl (+)
```

instead of

```
foldl (\x y -> x + y)
```

and also pointed out that I could simply write

```
[a..b]
```

instead of

```
[n | n <- [a .. b]]
```

He also suggested that I consider writing functions with guards, and wrote "Haskell syntax is like a box of chocolates…"

I responded:

> These are great suggestions, thanks!
>
> I guess I was writing a lambda to illustrate the parallels of addition to my string accumulation, but certainly if I really wanted "+" I should just use the existing function.
>
> Short function names — check. I have gotten used to writing quite long method names in C++ (for classes with a lot of methods). It has probably instilled habits that are not helpful in Haskell!
>
> Jason, I appreciate your rewrite… it is definitely improved.
>
> I will prepare a followup presenting an improved version.

An unknown blogger user let me know that it was possible to use descending sequences if I gave GHCi a hint that I wanted it to count down, instead of up: that is, GHCi would not accept

```
[5..1]
```

but it *would* accept

```
[5,4..1]
```

An unknown blogger also wrote:

> Also, folding a [**Bool**] into a successively concatenated **String** doesn't seem ideal. Since **String** is equivalent to [**Char**], you could map each **Bool** directly to its equivalent **Char**:

```
$ ghci
Prelude> let stringify_vals = map (\x -> if x then '#' else ' ')
Prelude> stringify_vals [False, False, True, False]
" # "
```

> If, however, your purpose was to demonstrate **fold**, then the version given is OK.
>
> Also also, I'm pretty sure most of the **__rot** functions aren't strictly necessary. You should be able to transpose the [String] right before **unlines**-ing it. So you would only need the two **prettyprint_blah_rot** functions.
>
> Also also also, I tried saving the body text as a **.lhs**, but GHCi choked on lines that started with '#'.
>
> Also x 4 (*sigh*), short names are a matter of taste. I wouldn't dare discourage using more descriptive names!
>
> Anyway, happy Haskelling!

Version 2

You can find version 2 of the code, the latest published version, here, in the form of a **.lhs** file. It is also included below.

```
The Dot Matrix Printhead: A Toy to Learn a Little Haskell
(and study the binary representation of two's-complement
numbers)

by Paul R. Potts

The author would like to thank several people who posted
comments on the original version of the code, especially
Jason Creighton, who made many improvements.

>import Data.Bits

Functions to generate lists of signed and
unsigned integers of numbits length:
```

```
>unsignedInts :: Int -> [Int]
>unsignedInts bits | (1 <= bits) && (bits <= 16) =
>  [(0::Int)..(bound::Int)]
>  where bound = 2 ^ bits - 1
>unsignedInts bits =
>  error "expected a value from 1 to 16"

>signedInts :: Int -> [Int]
>signedInts bits | (1 <= bits) && (bits <= 16) =
>  [(-(bound::Int)) - 1..(bound::Int)]
>  where bound = 2 ^ (bits - 1) - 1
>signedInts bits =
>  error "expected a value from 1 to 16"
```

The following code maps the binary representations of numbers into printable characters.

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

The integer generating function (signed or unsigned) comes in as a parameter; that's what (Int -> [Int]) means. We next get an integer argument (the number of bits), and finally, we spit out a list of characters.

```
>bitPicture :: (Int -> [Int]) -> Int -> [Char]

>bitPicture intGen bits =
>  unlines $ map stringAtIndex (reverse [0..(bits-1)])
>  where
>    stringAtIndex = map boolToChar . bitsAtIndex
>    bitsAtIndex idx = [testBit n idx |
>      n <- intGen bits]
```

Note that the "where" clauses allow us to set up some simplifying expressions. Let's look at the first one:

```
StringAtIndex = map boolToChar . bitsAtIndex
```

OK, this is slightly mind-twisting right off the bat. In the world of C and Java you can't write things like that. The dot is function composition expressed naturally; instead of something like

```
stringOfBoolsFromBitsAtIndex idx =
    map boolToChar (bitsAtIndex idx)
```

we can just get Haskell to chain the functions
together for us. No fuss, no muss. But this function
makes a forward reference to bitsAtIndex, which we
haven't seen yet. This forward reference to a function
we haven't defined yet would be completely illegal in
C, although there is "forward declaration" workaround.

Let's make sure we understand what bitsAtIndex does:

```
bitsAtIndex idx = [ testBit n idx | n <- intGen bits ]
```

To break down the list comprehension, the expression on
the right hand side of the vertical bar says that we feed
the supplied integer generator function the number of bits
to use. That gives us a list. From that list we will draw
our n values and. Instead of generating the bitmaps for
each integer value, like I did initially, and then using
"transpose" to rearrange them, this version accesses the
"columns" of the table directly! With these two expressions,
we've got our string representing all the bits at a
given index _for the whole supplied set of integers_.

This part is relatively simple:

```
bitPicture intGen bits =
    unlines $ map stringAtIndex (reverse [0..(bits-1)])
```

We already know what unlines does; it turns our list
of strings into a properly line-broken string. We've
just described how stringAtIndex works, and

```
map stringAtIndex (reverse [0..(bits-1)])
```

just feeds stringAtIndex a set of indices mapped from
high to low, so that our most significant bitcomes out
leftmost. But what does "$" do?

This is the "right-associating infix application operator
(f $ x = f x) useful in continuation-passing style."

I'm not going to go into the meaning of "continuation-
passing style" right now, but I it saves us from having
to write:

```
unlines $ map (stringAtIndex
    (reverse [0..(bits-1)]))

to get the right order of operations, which is nice.

Anyway, let's try it. We need to drive the function:

>main :: IO ()
>main = do
>   print "Unsigned values (6 bits)"
>   putStr $ bitPicture unsignedInts 6
>   print "Signed values (6 bits)"
>   putStr $ bitPicture signedInts 6

After using GHCi's :cd and :load command to read
this file, just type:

main

"Unsigned values (6 bits)"
                                ################################
                    ################                ################
            ########        ########        ########        ########
        ####    ####    ####    ####    ####    ####    ####    ####
    ##  ##  ##  ##  ##  ##  ##  ##  ##  ##  ##  ##  ##  ##  ##  ##  ##
    # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

"Signed values (6 bits)"
    ################################
                    ################                ################
            ########        ########        ########        ########
        ####    ####    ####    ####    ####    ####    ####    ####
    ##  ##  ##  ##  ##  ##  ##  ##  ##  ##  ##  ##  ##  ##  ##  ##  ##
    # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
```

Discussion on the Second Version (from Blogger Comments)

Neil Mitchell wrote:

> A couple of comments, just tiny refinements:

```
unsignedInts :: Int -> [Int]
unsignedInts bits | (1 <= bits) && (bits <= 16) =
 [(0::Int)..(bound::Int)]
 where bound = 2 ^ bits - 1
```

I'd write this as:

```
unsignedInts :: Int -> [Int]
unsignedInts bits | 1 <= bits && bits <= 16 = [0..bound]
where bound = 2 ^ bits - 1
```

The brackets are unnecessary.

I'm pretty sure the type signature is entirely redundant? The type sig on **unsignedInts** should make it all Ints. If you did need a type sig, I'd put it on bound:

```
where bound = (2 ^ bits - 1) :: Int
```

signedInts could be refined in a similar way. And

```
bitPicture intGen bits = unlines $ map stringAtIndex (reverse [0..(bits-1)])
 where
 stringAtIndex = map boolToChar . bitsAtIndex
 bitsAtIndex idx = [ testBit n idx | n <- intGen bits ]
```

could be written as

```
bitPicture intGen bits = unlines $ map stringAtIndex $ reverse [0..bits-1]
 where
 stringAtIndex = map boolToChar . bitsAtIndex
 bitsAtIndex idx = map (`testBit` n) (intGen bits)
```

No need for a list comp in the second one, and a few less brackets all round :)

Neil added:

Oh, and don't worry about **$** — it's useful in continuation-passing style, but it's a lot more useful to avoid some brackets!

```
f (big thing) == f $ big thing
```

```
f (big (thing here)) = f $ big $ thing here
```

I replied:

Hi Neil,

Thanks for your suggestions!

I think you are correct that there is no need for the explicit typing if a type signature is provided.

Although it seems to work correctly, the expression

```
1 <= bits && bits <= 16
```

(with no parentheses) makes me a little nervous. I work a lot C++, in which some operators have the wrong precedence. The above parses correctly in C++, but the expression

```
x & 1 == 0
```

doesn't say what you mean, unless you mean "always false," because "==" binds more tightly than bitwise "and." C programmers who endeavor to make life easier for their successors tend to fully parenthesize expressions that mix different kinds of operators. I don't really want to break that habit for Haskell or I'll be writing expressions that are silently incorrect when I go back to C/C++.

I have a similar problem with variable names; in Dylan, names like **gen-total** and ***module-var-*** are not just legal but conventional, and if you want the "-" and "*" to be interpreted as a separate token you have to leave spaces around them. When I go from Dylan back to C I tend to write illegal variable names!

The conversion of

```
bitPicture intGen bits = unlines $ map stringAtIndex (reverse [0..(bits - 1)])
```

to

```
bitPicture intGen bits = unlines $ map stringAtIndex $ reverse [0..(bits - 1)]
```

seems entirely good, using the **$** operator again; but the change from

```
bitsAtIndex idx = [ testBit n idx | n <- intGen bits ]
```

to

```
bitsAtIndex idx = map (`testBit` n) (intGen bits)
```

loses the definition for **n**, so I get

```
Not in scope: 'n'
```

I have played around a bit with other possible ways to get rid of the comprehension here but they did not seem to work right. In Scheme I would consider using **rcurry** to make **testBit idx** into a function I could **map** the list onto, but I am not quite sure how to get that effect in Haskell.

I also wrote:

Oh, I found what I was looking for… a way to use ".." to express a descending list. Instead of:

```
reverse [0..(bits - 1)]
```

I can write

```
[bits - 1, bits - 2..0]
```

Clearer? More efficient? I'm not sure.

Blogger user The Alternate Moebyus wrote:

> Paul, I'd say the comprehension is more efficient, as it just becomes **enumFromThenTo**, and does not involve reversing a list.

An unknown user wrote:

> Regarding:

```
bitsAtIndex idx = [ testBit n idx | n <- intGen bits ]
```

> changed to

```
bitsAtIndex idx = map (`testBit` n) (intGen bits)
```

> I think you might be misunderstanding things a little. **n** should be **idx**. Here, we'll start simple, using a lambda to simulate the list comprehension:

```
bitsAtIndex idx = map (\n -> testBit n idx) (intGen bits)
```

> Then we'll use the standard function **flip** to flip the arguments:

```
bitsAtIndex idx = map (\n -> flip testBit idx n) (intGen bits)
```

> Now, by the definition of lambda, we know that

```
(\x -> f x)
```

> is equivalent to

```
(f)
```

> so we remove the lambda (in this case **f** is **(flip testBit idx)**):

```
bitsAtIndex idx = map (flip testBit idx) (intGen bits)
```

> This could also be written:

```
bitsAtIndex idx = map (`testBit` idx) (intGen bits)
```

> since

```
(x `f` y)
```

> is equivalent to

```
(f x y)
```

> Figuring out why is left as a learning experience. :)

I wrote:

> Thanks for the ongoing advice!
>
> I had a conversation with another Paul here in Ann Arbor who is a Ph.D. student in Computer Science. It turns out Haskell is his favorite language, so we had an interesting and helpful discussion on currying, and he mentioned **flip**. Of course, everyone else at that get-together quickly found other people to talk to! I have also been studying Bird's book, which has interesting material on list comprehensions. I'm getting there…

and:

> Some further followup:
>
> GHC is confused by Literate Haskell programs that have lines starting with #. The Haskell 98 report does not mention anything special about the # character, but a bug report about this issue on the GHC site indicates it has something to do with special support for C preprocessor directives and will not likely be changed. I remembered to indent the output lines starting with # in my first article, but not the second. I have fixed that and will try to remember to test the **.lhs** content as-is in the future.
>
> Neil, you are right that the line

```
bitsAtIndex idx = map (`testBit` idx) (intGen bits)
```

> works. I should have been able to figure that out. I thought I had tried making that change but I must have broken something else and just made myself more confused. I have also followed your steps and feel like I still understand, so I think that's a good sign!

Neil Mitchell replied:

> Glad you figured it out — sorry for the one character typo in the original :)
>
> I did consider suggesting

```
[bit-1,bit-2..0]
```

> instead of the **reverse**, but it doesn't look as clean. It would be possible to add a GHC rule to convert:

```
reverse (fromEnumTo ...)
```

> to switch the arguments, which is always safe, and would give better performance, without having to duplicate (**bit**).
>
> I understand your comments about trying to keep "in C mindset" as well, it can get very confusing. I tend to either type semi-colons

in Haskell, or miss them in C, depending on which transition I'm making.

Followup to the Discussion of the Second Version

For those following along the discussion in the comments, here is an alternate implementation of the **bitPicture** function:

```
bitPicture intGen bits =
    unlines $ map stringAtIndex $
    [bits - 1, bits - 2..0]
    where stringAtIndex = map boolToChar . bitsAtIndex
        bitsAtIndex idx = map (flip testBit idx)
            (intGen bits)
```

Note that using [**bits - 1, bits - 2..0**] in the first line allows us to avoid the list reversal, so that seems like a clear win. The method of expressing **bitsAtIndex** above is one of several that were suggested. The idea behind the **flip** is that the return value of **intGen** should become the first parameter to **bitsAtIndex**, not the second. You can also express this by forcing **stringAtIndex** to be treated as an infix operator:

```
bitsAtIndex idx = map (`testBit` idx) (intGen bits)
```

Both of these alter the behavior of Haskell's automatic currying so that it replaces the second parameter instead of the first, leaving the first as the sole parameter to the resulting curried function; in Common Lisp, you might do this explicitly using **rcurry**. And don't forget the original, using a list comprehension:

```
    bitsAtIndex idx = [testBit n idx | n <- intGen bits]
```

Assuming there is not a specific concern about efficiency, and there certainly isn't for this toy program, it seems like the choice between these versions might be a matter of style. Thanks again to everyone who made suggestions.
