99 Bottles of Beer

Kiyoshi Akima
http://kiyoshiakima.tripod.com/funprogs

2006.04.27

Contents

1 The Song

Okay, you all know the song. You sang it to while away the hours while riding the bus to summer camp. Or in the back of your parents' car on the way to the beach. Admit it, I know you sang it.

> Ninety-nine bottles of beer on the wall,
> Ninety-nine bottles of beer.
> Take one down, pass it around,
> Ninety-eight bottles of beer on the wall.
>
> Ninety-eight bottles of beer on the wall,
> Ninety-eight bottles of beer.
> Take one down, pass it around,
> Ninety-seven bottles of beer on the wall.
> ...
>
> Two bottles of beer on the wall,
> Two bottles of beer.
> Take one down, pass it around,
> One bottle of beer on the wall.
>
> One bottle of beer on the wall,
> One bottle of beer.
> Take one down, pass it around,
> No more bottles of beer on the wall.

It shouldn't take much to write a computer program to generate all the verses; after all, computers are supposed to be good at this sort of mindless, repetitive drudgery.

No, it shouldn't take much, and it doesn't. It's been done. Done a *lot*.

There's a web site called 99 Bottles of Beer that collects and showcases versions written in various programming languages. As of this writing there are over nine hundred versions available. It may be over a thousand by the time you read this. (The URL is `http://www.99-bottles-of-beer.net` .)

So who has more time to waste? A person who sings the song or a person who writes a program to generate the verses? (Or is it the people who maintain the above-mentioned site?)

I don't mean to poke fun at the site. After all, I'm spending time writing this document. And the site does showcase an incredible variety of languages and programming styles, ranging from the programming language for the Babbage Analytical Engine (yes, there's an emulator available in case you want to run this program) to the latest cutting-edge web-based languages. C++ versions include obfuscated, object-oriented, and extreme template meta-programming versions. There are even programs shaped like beer bottles.

2 A Program for the HP 48

It doesn't even take a computer to generate the verses. Jeromy Carriere has written a program for the HP 48 programmable calculator and contributed it to the collection at 99 Bottles of Beer. With minor formatting changes, it is reproduced here:

```
«
  10 1 FOR k
  IF k 1 == THEN
    "1 bottle of beer on the wall."
    "1 bottle of beer."
  ELSE
    " bottles of beer on the wall."
    1 k →STR REPL
    " bottles of beer."
    1 k ->STR REPL
  END
  "Take one down and pass it around."
  IF k 1 == THEN
    "No more bottles of beer on the wall."
  ELSE
    IF k 2 == THEN
      "1 bottle of beer on the wall."
    ELSE
      " bottles of beer on the wall."
      1 k 1 - →STR REPL
    END
  END
  -1 STEP
»
```

As written, this program generates the last ten verses and leaves the strings on the stack. A simple change makes it generate all 99 verses, albeit at a cost in time and stack space.

3 A Program in System RPL

This program is my contribution to the collection. While also intended for the HP 48, this one is written in System RPL (SysRPL). SysRPL is an intermediate-level language between UserRPL (programs entered from the keyboard) and the machine language understood by the processor.

3a ⟨*beer.s* 3a⟩≡ 3b▷

```
( 99 Bottles of Beer   )
( HP 48 SysRPL Version )
( by Kiyoshi Akima     )
(  k_akima@hotmail.com )
( 2006.04.24           )

ASSEMBLE
    NIBASC  /HPHP48-B/
RPL
```

The program consists of initialization, a main loop doing one verse at a time until it runs out of beer, and clean up.

3b ⟨*beer.s* 3a⟩+≡ ◁3a

```
::
    ⟨initialize 3c⟩
    BEGIN
        ⟨do one verse 5b⟩
    1GETLAM #0= UNTIL
    ⟨clean up 6d⟩
;
```

Unlike Jeromy's program, this program will display seven lines at a time on the screen. Each additional line will scroll the others up the screen, like a teleprompter (or the opening scene in *Star Wars* but without the 3-D perspective effects and without the soundtrack).

3.1 Initialization

Before going into the main loop, the program needs to initialize.

3.1.1 Stack

The program expects no arguments on the stack.

3c ⟨*initialize* 3c⟩≡ (3b) 3d▷

```
AtUserStack
```

3.1.2 Display

Before the program starts the main loop, it turns off the ticking clock (whether or not it's actually displayed) and turns off the menu.

3d ⟨*initialize* 3c⟩+≡ (3b) ◁3c 4a▷

```
RECLAIMDISP ClrDA1IsStat TURNMENUOFF
```

3.1.3 Output Strings

The calculator's screen has seven usable rows of text, and the program will use all of them. The program will scroll the text up the screen as it goes. While the system is capable of doing a smooth pixel-by-pixel scroll, this program does it line-by-line. (The interested reader is welcome to make this enhancement.)

4a ⟨*initialize* 3c⟩+≡ (3b) ◁3d 4b▷

```
NULL$ SEVEN NDUPN DROP
```

3.1.4 Scrolling

A secondary will scroll the display and add a new line at the bottom.

4b ⟨*initialize* 3c⟩+≡ (3b) ◁4a 4g▷

```
' ::
    ⟨scroll display 4c⟩
;
```

First, the six lines at the bottom are moved up one line.

4c ⟨*scroll display* 4c⟩≡ (4b) 4d▷

```
3GETLAM 4GETLAM 5GETLAM 6GETLAM 7GETLAM 8GETLAM
9PUTLAM 8PUTLAM 7PUTLAM 6PUTLAM 5PUTLAM 4PUTLAM
```

Then the new line is added at the bottom.

4d ⟨*scroll display* 4c⟩+≡ (4b) ◁4c 4e▷

```
3PUTLAM
```

And finally the display is redrawn.

4e ⟨*scroll display* 4c⟩+≡ (4b) ◁4d

```
3GETLAM  4GETLAM  5GETLAM  6GETLAM  7GETLAM  8GETLAM  9GETLAM
DISPROW1 DISPROW2 DISPROW3 DISPROW4 DISPROW5 DISPROW6 DISPROW7
```

If you want to slow down the program, this is one possible place to insert a delay loop. One such delay loop is presented here. This loop takes a square root once for each bottle of beer on the wall. Note that this causes the delay to get shorter and shorter as the program progresses.

4f ⟨*possible delay loop* 4f⟩≡

```
1GETLAM UNCOERCE
1GETLAM ZERO_DO (DO)
    %SQRT
LOOP
DROP
```

3.1.5 Number of Bottles

And, of course, the program needs to keep track of the number of bottle(s) of beer on the wall.

4g ⟨*initialize* 3c⟩+≡ (3b) ◁4b 5a▷

```
99
```

3.1.6 Memory Allocation

The seven display strings, the scroll secondary, and the number of bottles are placed in a null-named temporary environment for easy access. A null-named temporary requires the program to access the elements by position instead of by name, but no memory is required for names and given the small number of elements, the burden on the programmer is minimal.

The program possibly could be made smaller and/or faster if the seven strings were put into a single list instead of seven separate variables. But this is the way I thought of it when I started writing the program and so this is the way it is.

By the way, it's no coincidence that the secondary is the second item on the stack. There just happens to be the word `2GETEVAL` which does the equivalent of `:: 2GETLAM EVAL ;`.

5a ⟨*initialize* 3c⟩+≡ (3b) ◁4g

```
NULLLAM NINE NDUPN {}N BIND
```

3.2 Doing One Verse

This is the heart of the program. Each iteration of the loop cranks out one verse, taking down one bottle of beer and handing it around.

3.2.1 First Line

First, the first line. Due to size constraints, this line is broken into two portions: "nn `bottle(s) of beer`" and " `on the wall,`" A copy of the first portion is left on the stack for the second line.

5b ⟨*do one verse* 5b⟩≡ (3b) 5d▷

```
1GETLAM #>$
⟨build "bottle(s) of beer" 5c⟩
DUP 2GETEVAL
" on the wall," 2GETEVAL
```

The rules of correct grammar require proper handling of the singular case.

5c ⟨*build "bottle(s) of beer"* 5c⟩≡ (5b 6c)

```
" bottle" &$
1GETLAM #1<> IT :: CHR_s >T$ ;
" of beer" &$
```

3.2.2 Second Line

Then the second line. Much of it was already built for the first line and left on the stack.

5d ⟨*do one verse* 5b⟩+≡ (3b) ◁5b 6a▷

```
CHR_. >T$ 2GETEVAL
```

3.2.3 Third Line

Then the third line. Again due to size constraints, this line is broken into two portions: "`Take one down,`" and "` pass it around.`"

6a ⟨*do one verse* 5b⟩+≡ (3b) ◁5d 6b▷

```
"Take one down," 2GETEVAL
" pass it around." 2GETEVAL
```

And the program fits its actions to its words, reducing the number of bottles on the wall by one.

6b ⟨*do one verse* 5b⟩+≡ (3b) ◁6a 6c▷

```
1GETLAM #1- 1PUTLAM
```

3.2.4 Fourth Line

And finally the fourth line. And again due to size constraints, this line is broken into two portions just like the first line.

The line built here could *almost* be used as the first line of the next verse, the only difference being the punctuation mark at the end. However, that would be *too* much like cheating. And it's not as if we're after maximum speed and/or minimum memory.

6c ⟨*do one verse* 5b⟩+≡ (3b) ◁6b

```
1GETLAM #0=ITE
    "No more"
    :: 1GETLAM #>$ ;
⟨build "bottle(s) of beer" 5c⟩
2GETEVAL
" on the wall." 2GETEVAL
```

3.3 Clean Up

Before terminating, the program needs to restore the display and free allocated memory.

3.3.1 Display

When the program is finished, it turns the menu back on and relinquishes the display, telling the system to update it.

6d ⟨*clean up* 6d⟩≡ (3b) 6e▷

```
TURNMENUON RECLAIMDISP ClrDAsOK
```

3.3.2 Memory

The program also needs to dispose of the temporary environment.

6e ⟨*clean up* 6d⟩+≡ (3b) ◁6d

```
ABND
```

4 Comparing the Programs

Given that the two programs don't do the same thing (their modes of output are completely different), it's not easy (nor necessarily meaningful) to compare them. However, here goes.

4.1 Size

The source code for my program is about twice the size of Jeromy's. But that's not what really matters. Jeromy's program is a little over 400 bytes in size, while mine is a little under 400, a ten-percent difference. Given that an `HP 48GX` has 128K of RAM, this hardly seems significant.

My program requires memory for the seven displayed strings and the next one as it's being built. Jeromy's program takes *much* more memory to run because it leaves all the strings on the stack. Still, even for all 99 verses, it's only about a tenth of the total memory of an `HP 48GX`.

4.2 Speed

This is where the two programs really differ. For all 99 verses, Jeromy's takes about three seconds while mine takes more than sixty. This is not because SysRPL is slower than UserRPL (quite the contrary), but because of the difference in output modes. In order to actually *see* the verses generated by Jeromy's program, you have to scroll through the stack. My program displays every line of every verse on the screen, though probably faster than you can read it—it's certainly faster than *I* can read.

Just for fun, I hacked Jeromy's program to display its results in a fashion similar to mine. I didn't bother to break up the long lines, so it only produced four lines for each verse instead of seven. Even so it took nearly twice as long as mine.

4.3 Development Time

I was able to get my program running in a couple of hours on a lazy Saturday afternoon (writing the rest of this document took somewhat longer). I don't know how long it took Jeromy to write his program, but I'm sure it was significantly less.

4.4 Conclusion

This comparison is not intended to show that one program is better than the other. They both do the job with a reasonable amount of resources and in adequate time (given the platform). Rather, I hope it points out some of the differences between two of the available programming languages for the `HP 48`, as well as illustrating two different approaches to solving the same problem.

Anyone care to try it in the Saturn assembly language? If you do, please submit it to the 99 Bottles of Beer site.

5 Building the Program

Before this program can be run on the calculator, it must first be built.

5.1 Literate Programming

This document not only describes the implementation of Ninety-nine Bottles of Beer, it *is* the implementation. The noweb system for "literate programming" generates both the document and the program code from a single source. This source consists of interleaved prose and labelled *Code Fragments*. The fragments are written in the order that best suits describing the program, namely the order you see in this document, not the order dictated by the programming language. The program noweave accepts the source and produces the document's typescript, which includes all of the code and all of the text. The program notangle extracts all of the code, in the proper order for compilation.

Fragments contain source code and references to other fragments. Fragment definitions are preceded by their labels in angle brackets. Several fragments may have the same name; notangle concatenates their definitions to produce a single fragment. noweave identifies this concatenation by using $+\equiv$ instead of $\equiv$ in continued definitions:

Fragment definitions are like macro definitions; notangle extracts a program by expanding one fragment. If its definition refers to other fragments, they themselves are expanded, and so on.

Fragment definitions include aids to help readers navigate among them. Each fragment name ends with the number of the page on which the fragment's definition begins and a letter giving its sequence within that page. If there is only one fragment on a page then there is no letter. This is also shown in the left margin. Each continued definition also shows the previous definition, and the next continued definition, if there is one. ◁ 7b is an example of a previous definition that appears on page 7, and 11 ▷ says the definition is continued on page 11. These annotations form a double linked list of definitions; the left arrow points to the previous definition in the list and the right arrow points to the next one. The previous link on the first definition is omitted, and the next link on the last definition is omitted. These lists are complete: If some of a fragment's definition appears on the same page with each other, the links refer to the page on which they appear.

Most fragments also show a list of pages on which the fragment is used. These unadorned use lists are omitted for root fragments, which define modules.

Of course, the simple fact that the documentation can be placed right next to the code doesn't necessary mean that the documentation and the code match, nor that either is any good. Still, it's a good start...

For more information about the noweb system of literate programming, please refer to `http://www.eecs.harvard.edu/~nr/noweb`.

5.2 Tangling

The following command line will extract the program source from the noweb document:

```
notangle -Rbeer.s -t4 beer48.nw >beer.s
```

This will create the file `beer.s`.

5.3 Compiling

Once the program source has been extracted, it can be compiled with the following command line.

```
rplcomp beer.s beer.a
```

This will produce an assembly file `beer.a`.

5.4 Assembling

The assembly file can then be assembled with the following command line:

```
sasm beer.a
```

This will produce an object file `beer.o` and a listing file `beer.l`.

5.5 Linking

The linker requires a command file:

9 ⟨*beer.m* 9⟩≡

```
LL beer.lr
OU beer
RE beer.o
SE c:\hp48\lib\entries.o
SU XR
```

This linker command file can be extracted from the noweb document with the following command line:

```
notangle -Rbeer.m beer48.nw >beer.m
```

This will create the file `beer.m`. (You will probably have to edit the location of `entries.o` to match your directory structure.) Then the linker can be run with this command line:

```
sload -H beer.m
```

If all goes well, this will produce an `HP 48` binary `beer` and a listing file `beer.lr`.

The binary can then be transferred to the `HP 48`, stored in a variable, and executed like any other program.

For further details on the building process, please consult the appropriate HP documentation.

A Index of Code Fragments

Underlined entries are to the definition of the Code Fragment. In many cases, the definition of a fragment can be continued from one piece to another.