Exhibiting Randomness in Arithmetic using Mathematica & C

G. J. Chaitin

chaitin@watson.ibm.com

In my book *Algorithmic Information Theory* I explain how I constructed a million-character equation that proves that there is randomness in arithmetic. My book only includes a few pages from the monster equation, and omits the software used to construct it. This software has now been rewritten in *Mathematica*.

The *Mathematica* software for my book, and its input, are here in their entirety. The *Mathematica* code is remarkably compact, but it sometimes is slow. So one *C* program plus equipment for automatically generating another is also included in this software package.

I used Version 2.1 of *Mathematica* as described in the second edition of Wolfram's book *Mathematica—A System for Doing Mathematics by Computer,* running on an *IBM RISC* System/6000 workstation.

Since the *APL2* character set is not generally available, I decided to change the symbols that denote the primitive functions in the toy *LISP* that I use in *Algorithmic Information Theory.*

There are seven different kinds of files:

- Included in this distribution:

  1. `*.m` files are *Mathematica* code.
  2. `*.c` files are *C* code.
  3. `*.lisp` files are toy *LISP* code. These are the four *LISP* programs in my book (`eval.lisp`, `eval2.lisp`, `eval3.lisp`, and `omega.lisp`), plus `test.lisp`.

4. `*.rm` are register machine code.

- These will produce:

5. `*.xrm` files are expanded register machine code (lower level code than that in `*.rm` files).
6. `*.run, *.2run, *.srun, *.mrun, *.crun, *.cmrun` files are the output from *LISP* interpreter runs.
7. `*.eq` files are exponential diophantine equations.

Six different *LISP* interpreters are included here:

1. `lisp.m` is a *LISP* interpreter written in nonprocedural *Mathematica* that uses *Mathematica* list structures to represent *LISP* S-expressions. Bindings are kept in a fast look-up table. `lisp.m` converts an `X.lisp` input file into an `X.run` output file.

$$\texttt{X.lisp} \longrightarrow \boxed{\texttt{lisp.m}} \longrightarrow \texttt{X.run}$$

2. `lisp2.m` is a *LISP* interpreter written in procedural *Mathematica* that uses *Mathematica* list structures to represent *LISP* S-expressions. Bindings are kept in a fast look-up table. `lisp2.m` converts an `X.lisp` input file into an `X.2run` output file.

$$\texttt{X.lisp} \longrightarrow \boxed{\texttt{lisp2.m}} \longrightarrow \texttt{X.2run}$$

3. `slisp.m` is a *LISP* interpreter written in procedural *Mathematica* that uses *Mathematica* character strings to represent *LISP* S-expressions. Bindings are kept in an association list that must be searched sequentially. `slisp.m` converts an `X.lisp` input file into an `X.srun` output file.

$$\texttt{X.lisp} \longrightarrow \boxed{\texttt{slisp.m}} \longrightarrow \texttt{X.srun}$$

4. `lispm.m` is a *Mathematica* program that simulates a *LISP* interpreter running on a register machine. `lispm.m` converts an `X.lisp` input file into an `X.mrun` output file.

   Before running this program, `xpnd.m` must be used to convert `lisp.rm` into `lisp.xrm`, which is needed by this program.

$$\begin{tikzcd} & \texttt{lisp.rm} \arrow[d] & \\ & \boxed{\texttt{xpnd.m}} \arrow[d] & \\ & \texttt{lisp.xrm} \arrow[d] & \\ \texttt{X.lisp} \arrow[r] & \boxed{\texttt{lispm.m}} \arrow[r] & \texttt{X.mrun} \end{tikzcd}$$

5. `clisp.m` is a *Mathematica* program serving as a driver for a *LISP* interpreter written in *C*. `clisp.m` converts an `X.lisp` input file into an `X.crun` output file.

   Before running `clisp.m`, the *C* program `lisp.c` must be compiled using the command `cc -O -olisp lisp.c`.

$$\begin{tikzcd} & \texttt{lisp.c} \arrow[d] & \\ & \boxed{\texttt{cc}} \arrow[d] & \\ & \texttt{lisp} \arrow[d] & \\ \texttt{X.lisp} \arrow[r] & \boxed{\texttt{clisp.m}} \arrow[r] & \texttt{X.crun} \end{tikzcd}$$

6. `clispm.m` is a *Mathematica* program serving as a driver for a *C* program that simulates a *LISP* interpreter running on a register machine. `clispm.m` converts an `X.lisp` input file into an `X.cmrun` output file.

   Before running `clispm.m`, `xpnd.m` must be used to convert `lisp.rm` into `lisp.xrm`. `rm2c.m` must then be used to convert `lisp.xrm` into the *C* program `lispm.c`. `lispm.c` is then compiled using the command `cc -O -olispm lispm.c`.

$$
\begin{tikzcd}
 & \texttt{lisp.rm} \arrow[d] & \\
 & \boxed{\texttt{xpnd.m}} \arrow[d] & \\
 & \texttt{lisp.xrm} \arrow[d] & \\
 & \boxed{\texttt{rm2c.m}} \arrow[d] & \\
 & \texttt{lispm.c} \arrow[d] & \\
 & \boxed{\texttt{cc}} \arrow[d] & \\
 & \texttt{lispm} \arrow[d] & \\
\texttt{X.lisp} \arrow[r] & \boxed{\texttt{clispm.m}} \arrow[r] & \texttt{X.cmrun}
\end{tikzcd}
$$

To run any one `X.m` of these six *LISP* interpreters, first enter *Mathematica* using the command `math`. Then tell *Mathematica*,

```
<< X.m
```

To run a *LISP* program `X.lisp`, enter

```
run @ "X"
```

To run several programs, enter

```
run /@ {"X","Y","Z"}
```

Before changing to another *LISP* interpreter, type `Exit` to exit from *Mathematica*, and then begin a fresh *Mathematica* session.

Here is how to run the *LISP* test program, the three *LISP* in *LISP* examples in my book, and then start computing the halting probability Ω in the limit from below:

```
math
<< clispm.m
```

```
run @ "test"
run /@ {"eval","eval2","eval3"}
Exit

math
<< clisp.m
run @ "omega"
Exit
```

The six different *LISP* interpreters run at vastly different speeds, but should always produce identical results. This can easily be checked, for example, as follows:

```
diff X.run X.crun > out
vi out
```

Two different front ends are available for these six *LISP* interpreters:

1. `run.m` is written in procedural *Mathematica*. As each M-expression is read in, it is written out, then converted to an S-expression that is written out and evaluated.[1]

2. `run2.m` is written in non-procedural *Mathematica*. All M-expressions are read in at once. Then each is converted to an S-expression that is written out and evaluated.

Which front end is used is determined by `frontend.m`. Each of the six *LISP* interpreters contains a `<<` of `frontend.m`. Normally `frontend.m` is `<< run.m` and the first front end is chosen. To select the second front end, change this to `<< run2.m`.

```
                                    << [run.m]
[LISP interpreter.m] << [frontend.m]
                                    << [run2.m]
```

Three register machine programs `*.rm` are provided: `example.rm`, `test.rm`, and `lisp.rm`. `example.rm` is the tiny example given in my Cambridge book. `test.rm` has each possible register machine instruction, but

[1] The conversion from M- to S-expression mostly consists of making all implicit parentheses explicit.

it is not a program that can be run. `lisp.rm` is the *LISP* interpreter used by `lispm.m` and `clispm.m`, and converted into the monster exponential diophantine equation by `eq.m`.

More precisely, to convert any one of the three register machine programs `X.rm` into an exponential diophantine equation there are two steps. First, use xpnd.m to convert `X.rm` into `X.xrm`. Then use `eq.m` to convert `X.xrm` into `X.eq`. For more output, set `fulloutput = True` before typing `<< eq.m`. For each conversion, a fresh copy of `eq.m` must be loaded into a clean *Mathematica* session.

$$\begin{tikzcd} & & & \texttt{fulloutput = True ?} \arrow[d] & \\ \texttt{X.rm} \arrow[r] & \boxed{\texttt{xpnd.m}} \arrow[r] & \texttt{X.xrm} \arrow[r] & \boxed{\texttt{eq.m}} \arrow[r] & \texttt{X.eq} \end{tikzcd}$$

Here is how to generate the monster equation:

```
 math
 << xpnd.m
 run @ "lisp"
 Exit

 math
[fulloutput = True]
 << eq.m
 fn of fn.xrm file = lisp
 Exit
```

How does this software help to exhibit randomness in arithmetic?

Take the equation in `lisp.eq`. Substitute 0 for `input[reg$X]` for each register `reg$X` except for `reg$expression`. Substitute a toy `LISP` expression that halts if and only if (the kth bit of the nth approximation to Ω is 1) for `input[reg$expression]`. (Most of the pieces for this are in `omega.lisp`.) The resulting exponential diophantine equation is $1. \times 10^6$ characters long and has $2. \times 10^4$ variables. It has exactly one solution for a given value of k and n if the kth bit of the nth approximation to Ω is 1. It has no solutions for a given value of k and n if the kth bit of the nth approximation to Ω is 0. Now think of n as a variable rather than as a parameter. The resulting equation has only finitely many solutions if the kth bit of Ω is 0. It has infinitely many solutions if the kth bit of Ω is 1.

Bibliography

[1] S. Wolfram, *Mathematica—A System for Doing Mathematics by Computer,* second edition, Addison-Wesley, 1991.

[2] B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* second edition, Prentice Hall, 1988.

[3] G. J. Chaitin, *Algorithmic Information Theory,* revised third printing, Cambridge University Press, 1990.

[4] G. J. Chaitin, *Information, Randomness & Incompleteness,* second edition, World Scientific, 1990.

[5] G. J. Chaitin, *Information-Theoretic Incompleteness,* World Scientific, 1992.

[6] G. J. Chaitin, "Randomness in arithmetic and the decline and fall of reductionism in pure mathematics," *IBM Research Report RC-18532,* November 1992.