Some Experiments on a Finite Machine

(incomplete working version)

C. Merck

December 17, 2008

Abstract

An otherwise universal computer model running within a finite memory space is employed in the experimental study of the halting probability, program-size complexity, maximum output length, and space complexity. The behavior of such a resource-limited machine is shown to be in principle distinct from that of the traditional Turing-machine. All source code used in conducting these experiments may be found in an appendix.

1 Motivation

The notion of a universal computer, famously developed by Alan Turing and Emmanuel Post in the 1930s (cite?), has formed the basis for the canonical theory of computation. However, the universal computer is not physically realizable, due to the requirement of infinite space.[1] Accordingly, present day physical computers, both those built by humans and those inherent in nature, are not universal but finite and as such are more accurately described by a finite theory than by the canonical theory of computation.

This work presents a particular finite machine and investigates its properties through experimentation on a PC. The software written to implement these experiments uses the Scheme programming language (PLT version), which lends itself particularly well to this application. All PC experiments performed require only a few minutes of computer-time on a modern machine, and so may be confirmed, modified, and retested with ease by the reader.

The intention in performing these experiments is that the results may be reconciled with a finite theory of computation and contrasted with the infinite machines and the canonical theory. Furthermore, a firm understanding of finite machines may prove useful in developing parallel quantum computers built up of many small, quite finite, machines.

[1] The terms *space*, *tape*, and *memory* are used interchangeably.

2 The Machine Model

Herein, experiments are performed on a finite machine which is programmed in an imperative structured programming language - a variant of the $\mathcal{P}''$ language developed by Böhm (cite?), referred to as $\mathcal{Q}$. A description of the function of $\mathcal{Q}$ is given here in this section, while a PC-based implementation in the SCHEME language may be found in an appendix.

2.1 The Programming Language

The basic operation of the machine is this: it begins reading the given program at the beginning, with all memory initialized to zero. The machine runs, reading instructions, manipulating memory, and writing to the output as directed by the program until reaching the programs end, resulting in a halt condition.

A $\mathcal{Q}$ program is a finite-length string of symbols from an alphabet of seven instructions. Each machine cycle the instruction indicated by the program pointer is read and action is taken depending on the instruction and the value of the memory cell indicated by the memory pointer. The effects of the seven instructions are:

$\mathcal{Q}$ instruction set	
$\mathfrak{f} = 0$	move memory pointer *forward* one cell, read next instruction
$\mathfrak{r} = 1$	move memory pointer *reverse* one cell, read next instruction
$\mathfrak{i} = 2$	*increment* cell at memory pointer, read next instruction
$\mathfrak{d} = 3$	*decrement* cell at memory pointer, read next instruction
$\mathfrak{w} = 4$	*write* cell value to end of output, read next instruction
$\mathfrak{l} = 5$	*loop* marker - no action, read next instruction
$\mathfrak{u} = 6$	*unless* cell at memory pointer is zero, return to matching $\mathfrak{l}$

The only restriction on input programs is that the loop instructions ($\mathfrak{l}$ and $\mathfrak{u}$) are matched. The set of syntactically correct $\mathcal{Q}$ programs is entirely described by the following grammar:

1. *Base Axiom:* The following are $\mathcal{Q}$ programs: $\mathfrak{f}$, $\mathfrak{r}$, $\mathfrak{i}$, $\mathfrak{d}$, $\mathfrak{w}$, and the empty string.
2. *Concatenation Rule:* Given any two $\mathcal{Q}$ programs $\vec{p}$ and $\vec{q}$, $\vec{p}$ & $\vec{q}$ is a $\mathcal{Q}$ program.
3. *Loop Rule:* Given any $\mathcal{Q}$ program $\vec{p}$, $\mathfrak{l}$ & $\vec{p}$ & $\mathfrak{u}$ is a $\mathcal{Q}$ program.

This grammar suggests a method of constructing syntactically correct programs, a challenge taken up in Section 2.4.

2.2 The Memory

The memory available to the machine is fixed during its operation, but may be chosen to be in various configurations. A particular machine, denoted by $\mathcal{Q}_m^n$,

has a fixed number, n, of memory cells (also known as *order*) each of which may be in any of m states (also known as *base*). More compactly, we say the memory is a vector in $\mathbb{Z}_m^n$. Consequently, when n and m are finite (which is hereafter assumed unless written otherwise), increment and decrement operations on the memory pointer and on any memory cell are taken (mod n) and (mod m) respectively. For example, attempting to increment a memory cell already at the maximum value of $m-1$ results in the cell's value becoming zero.

In the case of infinite order, $\mathcal{Q}$ is Turing-complete. The $\mathcal{P}''$ machine has been shown to be so, and every $\mathcal{Q}_\infty$ program may be translated into an equivalent $\mathcal{P}''$ program by the procedure given in Section **??**.

2.3 $\mathcal{Q}$ as a Recursive Function

Formally, $\mathcal{Q}$ is a function which maps program strings to output strings. When $\mathcal{Q}(\vec{p})$ converges given a particular program string $\vec{p}$, denoted $\mathcal{Q}(\vec{p})\downarrow$, the program is said to halt with output $\mathcal{Q}(\vec{p})$. Otherwise the program does not halt, that is, it diverges, denoted $\mathcal{Q}(\vec{p})\uparrow$.

We can write down the exact behavior of $\mathcal{Q}$ in a single recurrence relation, but first the initial conditions for that recurrence must be established. We let

$$\mathcal{Q}_m^n(\vec{p}) = \mathcal{Q}_m^n[0, \vec{r}_0, 0, \vec{s}_0](\vec{p}), \quad \text{where } \vec{r}_0 = \vec{0} \in \mathbb{Z}_m^n \text{ and } \vec{s}_0 \text{ is the empty vector}^2. \tag{1}$$

Now we can define the recurrence relation describing the machine's operation:

$$\mathcal{Q}_m^n[i, \vec{r}, j, \vec{s}](\vec{p}) = \tag{2}$$

$$\begin{cases} \vec{s}, & \text{if } i = |\vec{p}| \\ \mathcal{Q}_m^n[i+1, \vec{r}, j+1 \pmod n, \vec{s}](\vec{p}), & \text{if } p^i = \mathfrak{f} \\ \mathcal{Q}_m^n[i+1, \vec{r}, j-1 \pmod n, \vec{s}](\vec{p}), & \text{if } p^i = \mathfrak{r} \\ \mathcal{Q}_m^n[i+1, \&_{i=0}^{n-1} \begin{cases} r^i + 1 \pmod m & \text{if } i = j \\ r^i & \text{otherwise} \end{cases}, j, \vec{s}](\vec{p}), & \text{if } p^i = \mathfrak{i} \\ \mathcal{Q}_m^n[i+1, \&_{i=0}^{n-1} \begin{cases} r^i - 1 \pmod m & \text{if } i = j \\ r^i & \text{otherwise} \end{cases}, j, \vec{s}](\vec{p}), & \text{if } p^i = \mathfrak{d} \\ \mathcal{Q}_m^n[i+1, \vec{r}, j, \vec{s}\,\&\, r^j](\vec{p}), & \text{if } p^i = \mathfrak{w} \\ \mathcal{Q}_m^n[i+1, \vec{r}, j, \vec{s}](\vec{p}), & \text{if } p^i = \mathfrak{l} \\ \mathcal{Q}_m^n[i+1, \vec{r}, j, \vec{s}](\vec{p}), & \text{if } p^i = \mathfrak{u} \wedge r^j = 0 \\ \mathcal{Q}_m^n[\mho_{\vec{p}}(i-1, 1), \vec{r}, j, \vec{s}](\vec{p}), & \text{if } p^i = \mathfrak{u} \wedge r^j \neq 0 \end{cases}, \tag{2}$$

$$\text{where } \mho_{\vec{p}}(i, d) = \begin{cases} i+1, & \text{if } d = 0 \\ \mho_{\vec{p}}(i-1, d-1), & \text{if } p^i = \mathfrak{l} \\ \mho_{\vec{p}}(i-1, d+1), & \text{if } p^i = \mathfrak{u} \\ \mho_{\vec{p}}(i-1, d), & \text{otherwise} \end{cases}. \tag{3}$$

Note that if the input programs are ungrammatical, $\mathcal{Q}(\vec{p})$ may diverge because of the failure of a single evalutation of $\mho_{\vec{p}}$ to converge. For example, if a program $\vec{p}$ contains unmatched $\mathbf{u}$ instructions, then $\mho_{\vec{p}}(i,1) \uparrow$ for some i. This strongly suggests that $\mathcal{Q}(\vec{p}) \uparrow$ [3]. However, in an effort to make $\mathcal{Q}$ independant of the behavior of $\mho$ for programs with bad-syntax, we will from now on only discuss programs $\vec{p}$ which are in the set of $\mathcal{Q}$-programs defined by the grammar in 2.1. The unfortunate side-effect of so restricting the domain of $\mathcal{Q}$ is a complication of the task of enumerating programs, discussed presently.

2.4 Enumerating Grammatical Programs

In order to tackle the problem of program-size complexity we must obtain a Gödel numbering for $\mathcal{Q}$-programs. That is, we must find a one-to-one function $\# : \mathcal{Q}\text{-programs} \mapsto \mathbb{N}$. Further, it is desired that this function always assigns larger numbers to longer programs. Two such definitions of $\#$ are proposed here.

The obvious Gödel numbering method is to define a syntax checking function and use it to filter all possible program strings. Let the syntax checker function be defined as

$$\mathcal{S}(\vec{p}) = \begin{cases} 1 & \text{if } \vec{p} \in \mathcal{Q}\text{-programs} \\ 0 & \text{otherwise} \end{cases}. \tag{4}$$

Then let the Gödel numbering of all possible program strings grammatical and ungrammatical alike, the *naïve* numbering, be defined as

$$\#_{\text{all}}(\vec{p}) = \sum_{i=0}^{|p|-1} 7^i p^i. \tag{5}$$

Now using the inverse of the above naïve Gödel numbering to give an ordered enumeration which is then filtered by the syntax checking function, we can define an acceptable numbering for grammatical programs:

$$\#(\vec{p}) = \begin{cases} \sum_{i=0}^{\#_{\text{all}}(\vec{p})} \mathcal{S}(\#_{\text{all}}^{-1}(i)) & \text{if } S(\vec{p}) \\ \text{undefined} & \text{otherwise} \end{cases}. \tag{6}$$

TODO: Write up a Gödel numbering method which applies the grammar rules to augment a set of known-grammatical programs.

3 Program-size Complexity Experiments

TODO: Give definition of Program-size complexity, H(s).

[3] We cannot say with certainty that

$$\forall \vec{p} (\exists i \in \{0 \,..\, |\vec{p}| - 1\} \text{ with } p^i = \mathbf{u} \text{ such that } \mho_{\vec{p}}(i-1,1) \uparrow) \Rightarrow \mathcal{Q}(\vec{p}) \uparrow$$

because of trivial counter-examples such as $\vec{p} = \mathbf{u}$ or **ííílδuu** where the last clause of the piecewise recurrence relation for $\mathcal{Q}$ is never evaluated.

3.1 String Sorting

TODO: Analyize results of psc sorting of output strings for various orders and bases. Specifically, how does the machine order effect the sorting?

3.2 Measuring Randomness

TODO: Discuss normalized Chaitin randomness. Use plots to show correlations.

4 The Busy Beaver Function

The *busy beaver*[4] function $\Sigma_U(|\vec{p}|)$ denotes the largest output which a program of length $|\vec{p}|$ or less can produce on a computer U. On Turing-complete machines Σ grows faster than any recursive function [?]. On finite machines however there exists a recursive upper-bound on the busy beaver function. We derive this bound via the following argument.

4.1 A Recursive Upper-Bound on $\Sigma_{\mathcal{Q}}$

A finite machine with a finite program and finite memory (a finite *machine-tuple*) has a finite number of unique states which it can occupy. Furthermore, a machine is deterministic in that its next state can always be determined from its previous state. Lastly suppose there exists a special state, call it *halted*, marking the end of execution. We can now reason that if a machine were ever to reach the halted state, it must have passed through every other state either exactly once or not at all - for if a state were to be revisited, the program would loop indefinitely and never halt.[5] So at the very most, the machine may pass through every one of its states once before halting. Since at most one symbol may be written to the output during each state transition (instruction), the output may contain no more symbols than there are states in the machine.

A machine $\mathcal{Q}_m^n$ given program $\vec{p}$ has a memory pointer with n possible states, a program pointer with $|\vec{p}|$ possible states, and n memory cells which may each be in any one of m states. In total there are $nm^n\,|\vec{p}|$ states. Thus

$$\Sigma_{\mathcal{Q}_m^n}(|\vec{p}|) \leq nm^n\,|\vec{p}|\,. \tag{7}$$

Note that it is not remarkable that individual values of Σ are finite, only that we can write down an upper-bound which is recursive in program-size. What is remarkable is that the upper-bound given above is not only recursive but linear in program-size.[6]

[4] The Σ definition given here is due to Chaitin (cite?)

[5] This argument is similar to Nietsche's eternal recurrence argument [ref].

[6] Idea: What if we chose the order to vary with program size. Then the busy-beaver function would be recursively bounded and arbitrarily powerful programs may be written. Of course, this would not make $\mathcal{Q}$ Turing-complete since a particular program of finite length would still have only a finite amount of memory allocated to it and could not process arbitrarily large input.

4.2 Results

This conclusion applied to $\mathcal{Q}$-machines entails an upper bound on the execution time of any finite length program. Since no $\mathcal{Q}$ instruction appends more than one value to the output string, any bound on execution time translates to a bound on output length. Some values of the busy beaver function for several machine orders are given here:

TODO: Tabulate busy beaver function for several n, m, and p-size.

5 Computing the Halting Probability

TODO: Give definition of Ω. Further, explain how Ω is computable for finite machines (is it?).

5.1 Chaitin's Algorithm

TODO: Give a brief overview of Chaitin's algoritm for computing Ω.

5.2 Results

TODO: Plot Ω_i versus i for various orders and bases.

6 Finite Function Experiments

The experiments performed so far use programs with no input. We were primarily concerned with the complexity of the program's output strings. However, the complexity of functions, mappings of inputs into outputs, may also be measured. In this section, functions of the form $f : \mathbb{Z}_m \mapsto \mathbb{Z}_m$ and a manner of implementing them as programs on $\mathcal{Q}$ is investigated.

6.1 Implementing Functions on $\mathcal{Q}$

It is desired to implement functions of the form $f : \mathbb{Z}_m \mapsto \mathbb{Z}_m$ (hereafter called *functions of base* m) as $\mathcal{Q}$-programs. This is accomplished by appropriately padding programs with increment and output instructions so that the function's argument is placed in the first memory cell and the memory cell pointed to when the program halts is taken to be the function's output.

In terms of the definition given earlier for $\mathcal{Q}$, we can define a higher-order function $\mathcal{R}$ such that $\mathcal{R}_m^n[\vec{p}]$ is the function of base m corresponding to $\vec{p}$ (containing no $\mathfrak{w}$ instructions) and order n. Let

$$\mathcal{R}_m^n[\vec{p}](a) = \begin{cases} \mathcal{Q}_m^n(\vec{p}\,\&\,\mathfrak{w})^0 & \text{if } a = 0 \\ \mathcal{R}_m^n[\mathfrak{i}\,\&\,\vec{p}](a-1) & \text{otherwise} \end{cases}. \tag{8}$$

To ensure clarity one elementary example is given. Consider the base-3 successor function, call it f. Naturally we have $f(0) = 1$, $f(1) = 2$, and $f(2) = 0$.

The function f can be implemented by the program which is a single increment instruction, requiring but one cell of memory. This is written $\mathcal{R}_3^1[\mathfrak{i}] = f$, and is derived as follows:

$$\begin{aligned}
\mathcal{R}_3^1[\mathfrak{i}](0) &= \mathcal{Q}_3^1(\mathfrak{iw})^0 \\
&= (1)^0 \\
&= 1 \\
&= f(0), \\
\mathcal{R}_3^1[\mathfrak{i}](1) &= \mathcal{R}_3^1[\mathfrak{ii}](0) \\
&= \mathcal{Q}_3^1(\mathfrak{iiw})^0 \\
&= (2)^0 \\
&= 2 \\
&= f(1), \\
\mathcal{R}_3^1[\mathfrak{i}](2) &= \mathcal{R}_3^1[\mathfrak{ii}](1) \\
&= \mathcal{R}_3^1[\mathfrak{iii}](0) \\
&= \mathcal{Q}_3^1(\mathfrak{iiiw})^0 \\
&= (0)^0 \\
&= 0 \\
&= f(2), \\
\therefore \mathcal{R}_3^1[\mathfrak{i}] &= f.
\end{aligned}$$

6.2 Sets of Implementable Functions

It is useful to concider the set of functions which are implementable as $\mathcal{Q}$ programs in the above-mentioned manner. To denote the functions implementable on $\mathcal{Q}_m^n$ we write Φ_m^n. Or, in terms of the above-defined $\mathcal{R}$ notation, we let

$$\Phi_m^n = \{\text{base-}m \text{ function } f \mid \exists \vec{p} : \mathcal{R}_m^n[\vec{p}] = f\}. \tag{9}$$

Furthermore we adopt the following short-hand for unions of these sets:

$$\Phi_m = \bigcup_{n=1}^{\infty} \Phi_m^n, \tag{10}$$

$$\Phi^n = \bigcup_{m=1}^{\infty} \Phi_m^n, \tag{11}$$

$$\Phi = \bigcup_{m=1}^{\infty} \bigcup_{n=1}^{\infty} \Phi_m^n. \tag{12}$$

6.3 Space Complexity

The space complexity, or *neatness*, of a function is the number of memory cells required to compute that function. We can leverage the newly defined

$\mathcal{R}$ notation in defining the neatness[7] of a base-m function f as

$$N(f) = min(\{n \mid f \in \Phi_m^n\}) \tag{13}$$

This notion of neatness is simple but is not obviously computable. Suppose you claim that you have computed $N(f)$ and found it to be equal to 3. Most likely you claim this because you have found a program $\vec{p}$ such that $\mathcal{R}_m^3[\vec{p}] = f$. But then I ask: on what grounds do you say that there exists no program $\vec{q}$ such that $\mathcal{R}_m^2[\vec{q}] = f$? If you are right, then you cannot answer my question in finite time by exhaustively searching through programs, since you will never find such a program $\vec{q}$. Therefore, without some theoretical knowledge of an upper-bound on program-size, one cannot compute N for cases where $N(f) > 1$.

In an attempt to eliminate this problem, we define a more qualified version of neatness, called *H-neatness*. Let

$$\begin{aligned} N_H(f) = min(\{n \mid f \in \Phi_m^n \ \wedge \\ |\vec{p}| = min(\{|\vec{q}| \,|\exists n' : \mathcal{R}_m^{n'}[\vec{q}] = f\}) \\ \}). \end{aligned} \tag{14}$$

In words, H-neatness is not the least memory required to implement a function but rather the memory required by the shortest program implementing that function.[8]. If it turns out that $\forall f : N(f) = N_H(f)$ then we will know that the finite-time computation of N is possible. This possibility is explored in Section 6.4.4. TODO: CORRECT THIS PARAGRAPH

Tables of functions of bases 2, 3, and 4 and their corresponding (H-)neatnesses and neat implementations may be found in Appendix B.

6.4 Several Hypotheses involving $\mathcal{R}$

There is much to be learned about finite machines, especially with regard to space-complexity. Here follows several conjectures and associated proofs or counterexamples where they have been discovered.

6.4.1 Composition Lemma

$$\forall n : \forall f, g \in \Phi_m^n : f \circ g \in \Phi_m^n. \tag{15}$$

In words, the set of functions implementable on $\mathcal{Q}_m^n$ is closed under composition.

OUTLINE OF PROOF: After running an arbitrary program on $\mathcal{Q}$ the memory and memory pointer are in an arbitrary state. However, if the program is

[7] We say a function f is neater than a function g if $N(f) < N(g)$.

[8] The relationship between neatness and H-neatness is analogous to that between computational complexity and Bennett's logical depth [Svozil's ref 35]

appended with a so-called *housekeeping routine* [9], the memory cells can be all initialized to zero with the exception of the cell at the memory pointer. If another arbitrary program is then appended, it will behave exactly as if it were run alone (without the first program and housekeeping routine). Thus, by placing the housekeeping routine between the implementation of two functions, the second function runs with the first program's return value as it's argument, which is the definition of function composition.

6.4.2 Finite Completeness Theorem

$$\forall f \text{ of base } m : f \in \Phi_m. \tag{17}$$

In words, any (finite) function from $\mathbb{Z}_m$ to $\mathbb{Z}_m$ can be implemented as a $\mathcal{Q}$-program in $\mathcal{R}$ notation.

Proof is by construction. Implementations of three elementary functions are proven. Then it is demonstrated that any function may be constructed through composition of the elementary functions.

Assume an order, n, of 3 and a base, m, of 2 or greater[10]. Define the three elementary functions

$$R(x) = x + 1, \tag{18}$$

$$S(x) = \begin{cases} 1 & \text{if x=0} \\ 0 & \text{if x=1} \\ x & \text{otherwise} \end{cases}, \tag{19}$$

$$G(x) = \begin{cases} 1 & \text{if x=0} \\ 1 & \text{if x=1} \\ x & \text{otherwise} \end{cases}, \tag{20}$$

called *rotate*, *swap*, and *group* respectively. Furthermore, it can be shown that these functions are implementable by the following examples:[11]

$$R(x) = \mathcal{R}^3_m[\mathfrak{i}], \tag{21}$$

$$S(x) = \mathcal{R}^3_m[\mathfrak{dfdflifurd}], \tag{22}$$

$$G(x) = \mathcal{R}^3_m[\mathfrak{ldfdruflfduf}], \tag{23}$$

$$\therefore R, S, G \in \Phi^3_m$$

Now, an algorithm for the construction of any base-m function f is presented, during which the reader should keep foremost in her mind the list of values

[9] The *housekeeping routine* of order n is defined as

$$\vec{h} = (\&_{n-1}\mathfrak{fliu}) \,\&\, \mathfrak{f}. \tag{16}$$

[10] The assumtion of $n = 3$ can be weakened to $n \geq 3$ if a better swap function can be found which behaves correctly when $n > 3$

[11] The reader may confirm for herself the correctness of these implementations.

$f(0), f(1), \ldots, f(m-1)$. Let $\textsc{Free}(x)$ denote the x^{th} number in the set $\mathbb{Z}_m - f(\mathbb{Z}_m)$. Let $\textsc{Dup}(x)$ indicate whether $f(x)$ has already appeared in the list of values, so that $\sum_{i=0}^{x-1} \textsc{Dup}(i)$ is the number of duplicate values in the range of f for arguments less than x.[12] Let α be f but with duplicate values replaced with the next free value

$$\alpha(x) = \begin{cases} \textsc{Free}(\sum_{i=0}^{x-1} \textsc{Dup}(i)) & \text{if } \textsc{Dup}(x) = 1 \\ f(x) & \text{otherwise} \end{cases}. \tag{24}$$

This function α is a permutation of the base-m identity function and so may be obtained through repeated composition of the rotate and swap functions.[13] Likewise, f may be obtained by repeatedly composing α with R, S, and G.[14] Since R, S, and G are in Φ_m^3 and f is a composition of R, S, and G, by the Composition Lemma we have that f is in Φ_m^3, Q.E.D.

6.4.3 A Bound on (H-)Neatness

$$\exists \text{ a recursive upper-bound } B_N \text{ s.t. } \forall f \text{ of base } m : N(f) < B_N(m). \tag{25}$$

$$\exists \text{ a recursive upper-bound } B_{N_H} \text{ s.t. } \forall f \text{ of base } m : N_H(f) < B_{N_H}(m). \tag{26}$$

The bound on neatness is derived using the Finite Completeness Theorem. Since all functions may be constructed from the elementary functions GS, ZS, and ZR, and the elementary functions have a neatness of at most 3, all functions have a neatness of at most 3.

I do not yet have a proof of a bound on H-neatness however. TODO: Discover this proof.

6.4.4 Nonequivalence of Neatnesses

$$\neg\forall f : N(f) = N_H(f). \tag{27}$$

PROOF BY EXAMPLE: The shortest implementation of the function $f(x) = 0$ for any order is **f**, valid when $n \geq 2$. So $N_H(f) = 2$. However, an implementation also exists when $n = 1$, specifically **lru**. So $N(f) = 1 \neq N_H(f)$, Q.E.D.

[12] A suitable, purely symbolic definition would be

$$\textsc{Dup}(x) = \begin{cases} 1 & \text{if } \exists y : y < x \wedge f(x) = f(y) \\ 0 & \text{otherwise} \end{cases}.$$

Note that this does not treat the first occurrence of a duplicated value as a duplicate.

[13] This method for construction of α from R and S is easily described as a recursive function, but it is rather technical and computer source code is more instructive. The reader is encouraged to review the Scheme function `build-alpha` in the source code (AppendixD.2)

[14] See the Scheme function `build-f` in AppendixD.2

6.4.5 Significance of Primes

TODO: Investigate significance of prime bases/orders. I feel this may be interesting because of the special properties of prime-power moduli in number theory.

7 Other Machine Models

The above analysis and conclusions were made with respect to a particular finite machine model. Other models of finite machines exist which may be subjected to similar analysis. Of particular interest are Turing-machines with finite tapes and finite cellular automata.

A universal Turing-machine with its infinite tape replaced with a finite tape, perhaps "shaped" like a Möbius strip so that seeking beyond the last cell returns the tape-head to the first cell. The behavior of this setup is expected to be similar but more general than that of $\mathcal{Q}$ due to the isomorphism between $\mathcal{Q}$ given some program and a particular Turing-machine with a blank finite tape.

TODO: Elaborate.

TODO: Discuss a variation of $\mathcal{Q}$ where the program is loaded into and read from memory, called $\bar{\mathcal{Q}}$. Here the problem of improper grammar is again important: perhaps the standard behavior of $\mho$ could be used, but with the first parameter (the temporary program pointer) being taken (mod $|\vec{p}|$).

8 Conclusions

The well known principle of the recursive unsolvability of the halting problem, which holds true for the universal machines of canonical computational theory, is invalid when applied to finite machines. Consequences of the solvability of the finite machine halting problem include the computability of the busy beaver function and the halting probability for finite machines. Furthermore, for the machine and function-program isomorphism assumed here, all finite functions of a given base can be computed on a machine of that base within at most three cells of memory. This *finite completeness* stands in contrast to the famous incompleteness of infinite machines.

A Symbol List

$\mathcal{Q}$	the machine model used here
$\mathbb{Z}$	the set of integers
$\mathbb{Z}_m$	the set of integers (mod m)
Σ	the busy beaver function
$\wedge$	boolean AND
$\uparrow$	diverges
$\downarrow$	converges
$\vec{x}$	a vector (sometimes called a string)
x^i	the i^{th} component of $\vec{x}$, if defined
&	vector append operator [15]
$\in$	is a member of
$\circ$	function composition
f^n	function composed with itself n times
$\mathcal{R}$	interpretation of a program as a finite function

[15] Two vectors or a vector and a scalar may be appended by this operator. Also, the following notations are defined:

$$\&_{i=0}^{n}\vec{f}(i) = \vec{f}(0)\,\&\,\vec{f}(1)\,\&\,...\,\&\,\vec{f}(n),$$
$$\&_n\,\vec{x} = \vec{x}\,\&\,\vec{x}\,\&\,...\,\&\,\vec{x}\ (n \text{ times}).$$

B Functions and their Neat Implementations

Here are tabulated all functions of bases 2 and 3 along with their corresponding shortest implementations on $\mathcal{Q}$ machines with different orders. TODO: Point out the neatest and H-neatest implementations where possible.

FUNCTION TABLE FOR BASE 2, ORDER UP TO 4, AND PROG-LEN UP TO 5

id	func	N_H	programs of order n=1..4			
0	(0 0)	2	[+]	<	<	<
1	(1 0)	1	+	+	+	+
2	(u 0)	1	+[]	+[]	+[]	+[]
3	(0 1)	1	*	*	*	*
4	(1 1)	2	[+]+	>+	>+	>+
5	(u 1)	1	+[]+	+[]+	+[]+	+[]+
6	(0 u)	1	[]	[]	[]	[]
7	(1 u)	1	[]+	[]+	[]+	[]+
8	(u u)	2	[[]+]	>+[]	>+[]	>+[]

FUNCTION TABLE FOR BASE 3, ORDER UP TO 4, AND PROG-LEN UP TO 6

id	func	N_H	programs of order n=1..4			
0	(0 0 0)	2	[+]	<	<	<
1	(1 0 0)					
2	(2 0 0)					
3	(u 0 0)					
4	(0 1 0)	2		[>-]>-	[>-]>-	[>-]>-
5	(1 1 0)	2		[>+]>-	[>+]>-	[>+]>-
6	(2 1 0)	2		>[->]<	<[>+]<	<[>+]<
7	(u 1 0)					
8	(0 2 0)					
9	(1 2 0)	1	+	+	+	+
10	(2 2 0)					
11	(u 2 0)					
12	(0 u 0)					
13	(1 u 0)					
14	(2 u 0)					
15	(u u 0)	1	+[]	+[]	+[]	+[]
16	(0 0 1)					
17	(1 0 1)					
18	(2 0 1)	1	-	-	-	-
19	(u 0 1)					
20	(0 1 1)	2		[->-]<	>[+<]<	[->-]<
21	(1 1 1)	2	[+]+	>+	>+	>+
22	(2 1 1)	2		+[>-]<	+[>-]<	+[>-]<
23	(u 1 1)					
24	(0 2 1)					

```
25      (1 2 1) 2               [>-]<   [>-]<   [>-]<
26      (2 2 1) 2               [>+]<   [>+]<   [>+]<
27      (u 2 1)
28      (0 u 1)
29      (1 u 1)
30      (2 u 1)
31      (u u 1) 1       +[]+    +[]+    +[]+    +[]+
32      (0 0 2) 2               [>+]>+  [>+]>+  [>+]>+
33      (1 0 2) 2               >[+>]<  <[>-]<  <[>-]<
34      (2 0 2) 2               [>-]>+  [>-]>+  [>-]>+
35      (u 0 2)
36      (0 1 2) 1       *       *       *       *
37      (1 1 2) 2               -[>-]<  <+[>]<  <+[>]<
38      (2 1 2) 2               +[>+]<  <-[>]<  <-[>]<
39      (u 1 2)
40      (0 2 2) 2               [+>+]<  >[-<]<  [+>+]<
41      (1 2 2) 2               -[>+]<  -[>+]<  -[>+]<
42      (2 2 2) 2       [+]-    >-      >-      >-
43      (u 2 2)
44      (0 u 2)
45      (1 u 2)
46      (2 u 2)
47      (u u 2) 1       +[]-    +[]-    +[]-    +[]-
48      (0 0 u)
49      (1 0 u)
50      (2 0 u)
51      (u 0 u) 1       -[]     -[]     -[]     -[]
52      (0 1 u)
53      (1 1 u)
54      (2 1 u)
55      (u 1 u) 1       -[]+    -[]+    -[]+    -[]+
56      (0 2 u)
57      (1 2 u)
58      (2 2 u)
59      (u 2 u) 1       -[]-    -[]-    -[]-    -[]-
60      (0 u u) 1       []      []      []      []
61      (1 u u) 1       []+     []+     []+     []+
62      (2 u u) 1       []-     []-     []-     []-
63      (u u u) 2       [[]+]   >+[]    >+[]    >+[]


FUNCTION TABLE FOR BASE 4, ORDER UP TO 3, AND PROG-LEN UP TO 8
id      func    N_H     programs of order n=1..3
0       (0 0 0 0)       2       [+]             <               <
1       (1 0 0 0)       3                                       <[->]>+
```

2	(2 0 0 0)	3		>[>++>]<
3	(3 0 0 0)	3		<[+>]>-
4	(u 0 0 0)			
5	(0 1 0 0)	3		<[+>-]<
6	(1 1 0 0)			
7	(2 1 0 0)	2	>-[->]<	+[>-]>-
8	(3 1 0 0)			
9	(u 1 0 0)			
10	(0 2 0 0)	3		>+[>-]>+
11	(1 2 0 0)	3		+>[->+]<
12	(2 2 0 0)	3		+>[+<+]<
13	(3 2 0 0)	2	->+[+>]<	++[>+]>+
14	(u 2 0 0)			
15	(0 3 0 0)	3		[+>+>+]<
16	(1 3 0 0)			
17	(2 3 0 0)	2	>->[+>]<	+<-[+>]<
18	(3 3 0 0)	2	>++[+>]<	
19	(u 3 0 0)			
20	(0 u 0 0)			
21	(1 u 0 0)			
22	(2 u 0 0)			
23	(3 u 0 0)			
24	(u u 0 0)	3		[+[>-]>]
25	(0 0 1 0)	3		-<[+>-]<
26	(1 0 1 0)			
27	(2 0 1 0)			
28	(3 0 1 0)	3		<[-<-<]<
29	(u 0 1 0)			
30	(0 1 1 0)	2	[->++]>-	[-<++]>-
31	(1 1 1 0)	3		+>[+<]<
32	(2 1 1 0)			
33	(3 1 1 0)	2	>+[+>]>+	>[++<+]<
34	(u 1 1 0)			
35	(0 2 1 0)	2	[>-]>-	[>-]>-
36	(1 2 1 0)	2	[>++]>-	[<++]>-
37	(2 2 1 0)	2	[>+]>-	[>+]>-
38	(3 2 1 0)	2	>[->]<	<[>+]<
39	(u 2 1 0)			
40	(0 3 1 0)			
41	(1 3 1 0)			
42	(2 3 1 0)	3		>+>[<+]<
43	(3 3 1 0)	2	-[>-]>++	-[>-]>++
44	(u 3 1 0)			
45	(0 u 1 0)			
46	(1 u 1 0)			
47	(2 u 1 0)			

48	(3 u 1 0)				
49	(u u 1 0)				
50	(0 0 2 0)				
51	(1 0 2 0)				
52	(2 0 2 0)				
53	(3 0 2 0)				
54	(u 0 2 0)				
55	(0 1 2 0)	3		+>+[->]<	>[->+]<
56	(1 1 2 0)				
57	(2 1 2 0)				
58	(3 1 2 0)				
59	(u 1 2 0)				
60	(0 2 2 0)	3			>[+<+]<
61	(1 2 2 0)	3			[>->-]<
62	(2 2 2 0)	3			+>[++<]<
63	(3 2 2 0)	2		++[>-]>+	<-<[+<]<
64	(u 2 2 0)				
65	(0 3 2 0)	2		+[>+]>+	+[>+]>+
66	(1 3 2 0)				
67	(2 3 2 0)				
68	(3 3 2 0)				
69	(u 3 2 0)				
70	(0 u 2 0)				
71	(1 u 2 0)				
72	(2 u 2 0)				
73	(3 u 2 0)				
74	(u u 2 0)				
75	(0 0 3 0)	3			+<[->+]<
76	(1 0 3 0)	3			<[+<+<]<
77	(2 0 3 0)				
78	(3 0 3 0)				
79	(u 0 3 0)				
80	(0 1 3 0)				
81	(1 1 3 0)	3			<[+>-]>+
82	(2 1 3 0)				
83	(3 1 3 0)				
84	(u 1 3 0)				
85	(0 2 3 0)	2		>-[+>]<	<-[+>]<
86	(1 2 3 0)	1	+	+	+
87	(2 2 3 0)	3			<+[+>]<
88	(3 2 3 0)	3			<++[+>]<
89	(u 2 3 0)				
90	(0 3 3 0)				
91	(1 3 3 0)	2		[++>+]<	[++>+]<
92	(2 3 3 0)				
93	(3 3 3 0)	3			+>[-<]<

94	(u 3 3 0)				
95	(0 u 3 0)				
96	(1 u 3 0)				
97	(2 u 3 0)				
98	(3 u 3 0)				
99	(u u 3 0)				
100	(0 0 u 0)	3			>[[>+]>]
101	(1 0 u 0)				
102	(2 0 u 0)				
103	(3 0 u 0)				
104	(u 0 u 0)	1	+[++]	+[++]	+[++]
105	(0 1 u 0)				
106	(1 1 u 0)				
107	(2 1 u 0)				
108	(3 1 u 0)				
109	(u 1 u 0)				
110	(0 2 u 0)				
111	(1 2 u 0)				
112	(2 2 u 0)				
113	(3 2 u 0)				
114	(u 2 u 0)	2		+[+>+]<	+[+>+]<
115	(0 3 u 0)				
116	(1 3 u 0)				
117	(2 3 u 0)				
118	(3 3 u 0)				
119	(u 3 u 0)				
120	(0 u u 0)	2		[[+>]+>]	[[+<]+>]
121	(1 u u 0)				
122	(2 u u 0)				
123	(3 u u 0)				
124	(u u u 0)	1	+[]	+[]	+[]
125	(0 0 0 1)	3			[->->-]<
126	(1 0 0 1)	2		>++[->]<	
127	(2 0 0 1)	2		>+>[->]<	-<+[->]<
128	(3 0 0 1)				
129	(u 0 0 1)				
130	(0 1 0 1)				
131	(1 1 0 1)	3			++>[+<]<
132	(2 1 0 1)	2		+[>++]>-	>[-<++]<
133	(3 1 0 1)				
134	(u 1 0 1)				
135	(0 2 0 1)				
136	(1 2 0 1)				
137	(2 2 0 1)	3			-<-[->]<
138	(3 2 0 1)	3			>+>[->]<
139	(u 2 0 1)				

140	(0 3 0 1)	3			<[+>+]<
141	(1 3 0 1)	2		-[++>-]<	<-[++>]<
142	(2 3 0 1)	1	++	++	++
143	(3 3 0 1)	2		+[++>+]<	<+[++>]<
144	(u 3 0 1)				
145	(0 u 0 1)				
146	(1 u 0 1)				
147	(2 u 0 1)				
148	(3 u 0 1)				
149	(u u 0 1)				
150	(0 0 1 1)				
151	(1 0 1 1)	3			->[+<]<
152	(2 0 1 1)				
153	(3 0 1 1)	2		[++>-]<	[++>-]<
154	(u 0 1 1)				
155	(0 1 1 1)	3			>[+<]<
156	(1 1 1 1)	2	[+]+	>+	>+
157	(2 1 1 1)	3			<[->]>++
158	(3 1 1 1)	3			<-[<+<]<
159	(u 1 1 1)				
160	(0 2 1 1)				
161	(1 2 1 1)	3			>[->++]<
162	(2 2 1 1)	2		+[->++]<	+[->++]<
163	(3 2 1 1)	2		+[>-]<	+[>-]<
164	(u 2 1 1)				
165	(0 3 1 1)				
166	(1 3 1 1)	3			[>++>+]<
167	(2 3 1 1)	2		->+[>-]<	++<+[>]<
168	(3 3 1 1)	3			<[+<+]>-
169	(u 3 1 1)				
170	(0 u 1 1)				
171	(1 u 1 1)				
172	(2 u 1 1)				
173	(3 u 1 1)				
174	(u u 1 1)				
175	(0 0 2 1)	2		-[>-]>-	-[>-]>-
176	(1 0 2 1)				
177	(2 0 2 1)				
178	(3 0 2 1)				
179	(u 0 2 1)				
180	(0 1 2 1)	2		-[>++]>-	[-<-<-]<
181	(1 1 2 1)	3			[<-<-]<
182	(2 1 2 1)				
183	(3 1 2 1)				
184	(u 1 2 1)				
185	(0 2 2 1)	2		-[>+]>-	-[>+]>-

186	(1 2 2 1)	2	[->++]<	[->++]<
187	(2 2 2 1)	3		[-<++]<
188	(3 2 2 1)			
189	(u 2 2 1)			
190	(0 3 2 1)	2	[+>+>]<	[+<+>]<
191	(1 3 2 1)	2	[>-]<	[>-]<
192	(2 3 2 1)	2	[>++]<	[>++]<
193	(3 3 2 1)	2	[>+]<	[>+]<
194	(u 3 2 1)			
195	(0 u 2 1)			
196	(1 u 2 1)			
197	(2 u 2 1)			
198	(3 u 2 1)			
199	(u u 2 1)			
200	(0 0 3 1)			
201	(1 0 3 1)	2	+[>+]>++	+[>+]>++
202	(2 0 3 1)	3		<[-<++]<
203	(3 0 3 1)			
204	(u 0 3 1)			
205	(0 1 3 1)			
206	(1 1 3 1)	3		+<-[>+]<
207	(2 1 3 1)			
208	(3 1 3 1)			
209	(u 1 3 1)			
210	(0 2 3 1)			
211	(1 2 3 1)	3		+<+[>]<
212	(2 2 3 1)			
213	(3 2 3 1)			
214	(u 2 3 1)			
215	(0 3 3 1)			
216	(1 3 3 1)	3		<[+>+]>+
217	(2 3 3 1)	2	+>-[>+]<	++<-[>]<
218	(3 3 3 1)	3		>[>+>+]<
219	(u 3 3 1)			
220	(0 u 3 1)			
221	(1 u 3 1)			
222	(2 u 3 1)			
223	(3 u 3 1)			
224	(u u 3 1)			
225	(0 0 u 1)			
226	(1 0 u 1)			
227	(2 0 u 1)			
228	(3 0 u 1)			
229	(u 0 u 1)			
230	(0 1 u 1)			
231	(1 1 u 1)			

232	(2 1 u 1)				
233	(3 1 u 1)				
234	(u 1 u 1)	1	+[++]+	+[++]+	+[++]+
235	(0 2 u 1)				
236	(1 2 u 1)				
237	(2 2 u 1)				
238	(3 2 u 1)				
239	(u 2 u 1)				
240	(0 3 u 1)				
241	(1 3 u 1)				
242	(2 3 u 1)				
243	(3 3 u 1)				
244	(u 3 u 1)	2		+[+>+]>+	+[+>+]>+
245	(0 u u 1)				
246	(1 u u 1)				
247	(2 u u 1)				
248	(3 u u 1)				
249	(u u u 1)	1	+[]+	+[]+	+[]+
250	(0 0 0 2)	3			>-[>+]>-
251	(1 0 0 2)	2		+>-[->]<	++[>-]>-
252	(2 0 0 2)	3			->[+>+]<
253	(3 0 0 2)	3			->[+>-]<
254	(u 0 0 2)				
255	(0 1 0 2)				
256	(1 1 0 2)				
257	(2 1 0 2)	2		+[>+]>-	+[>+]>-
258	(3 1 0 2)				
259	(u 1 0 2)				
260	(0 2 0 2)	2		[+>++>]<	[+<++>]<
261	(1 2 0 2)				
262	(2 2 0 2)	3			>[<++]<
263	(3 2 0 2)				
264	(u 2 0 2)				
265	(0 3 0 2)				
266	(1 3 0 2)	3			>->[+>]<
267	(2 3 0 2)	3			+<+[+>]<
268	(3 3 0 2)				
269	(u 3 0 2)				
270	(0 u 0 2)				
271	(1 u 0 2)				
272	(2 u 0 2)				
273	(3 u 0 2)				
274	(u u 0 2)				
275	(0 0 1 2)	2		>+[->]<	<+[->]<
276	(1 0 1 2)	3			<++[->]<
277	(2 0 1 2)	3			<-[->]<

278	(3 0 1 2)	1	-	-	-
279	(u 0 1 2)				
280	(0 1 1 2)				
281	(1 1 1 2)	3			-[<-<-]<
282	(2 1 1 2)				
283	(3 1 1 2)	2		>+[>-]<	-<+[>]<
284	(u 1 1 2)				
285	(0 2 1 2)				
286	(1 2 1 2)				
287	(2 2 1 2)	3			+[-<++]<
288	(3 2 1 2)	2		+[>++]<	+[>++]<
289	(u 2 1 2)				
290	(0 3 1 2)				
291	(1 3 1 2)				
292	(2 3 1 2)				
293	(3 3 1 2)	3			-<-[>]<
294	(u 3 1 2)				
295	(0 u 1 2)				
296	(1 u 1 2)				
297	(2 u 1 2)				
298	(3 u 1 2)				
299	(u u 1 2)				
300	(0 0 2 2)	3			>[+>+]<
301	(1 0 2 2)	2		++[>+]>-	<+<[-<]<
302	(2 0 2 2)	3			->[++<]<
303	(3 0 2 2)	3			[>+>+]<
304	(u 0 2 2)				
305	(0 1 2 2)	2		[++>-]>+	+<-[->]<
306	(1 1 2 2)	2		-[->++]<	-[->++]<
307	(2 1 2 2)	3			-<[+>]>+
308	(3 1 2 2)				
309	(u 1 2 2)				
310	(0 2 2 2)	3			>[++<]<
311	(1 2 2 2)	3			<[+>]>+
312	(2 2 2 2)	2	[+]++	>++	>++
313	(3 2 2 2)	3			<[->]>-
314	(u 2 2 2)				
315	(0 3 2 2)	2		+[>-]>+	+[>-]>+
316	(1 3 2 2)				
317	(2 3 2 2)	3			[+<++]<
318	(3 3 2 2)	2		[+>++]<	[+>++]<
319	(u 3 2 2)				
320	(0 u 2 2)				
321	(1 u 2 2)				
322	(2 u 2 2)				
323	(3 u 2 2)				

324	(u u 2 2)				
325	(0 0 3 2)	2		[>+]>+	[>+]>+
326	(1 0 3 2)	2		>[+>]<	<[>-]<
327	(2 0 3 2)	2		[>-]>+	[>-]>+
328	(3 0 3 2)	2		[>++]>+	[<++]>+
329	(u 0 3 2)				
330	(0 1 3 2)				
331	(1 1 3 2)	2		-[>-]<	-[>-]<
332	(2 1 3 2)				
333	(3 1 3 2)				
334	(u 1 3 2)				
335	(0 2 3 2)				
336	(1 2 3 2)	2		-[>++]<	-[>++]<
337	(2 2 3 2)	3			<+<[>+]<
338	(3 2 3 2)				
339	(u 2 3 2)				
340	(0 3 3 2)				
341	(1 3 3 2)	2		-[>+]<	-[>+]<
342	(2 3 3 2)	2		>+>[>+]<	-[+>++]<
343	(3 3 3 2)	3			<+>[>+]<
344	(u 3 3 2)				
345	(0 u 3 2)				
346	(1 u 3 2)				
347	(2 u 3 2)				
348	(3 u 3 2)				
349	(u u 3 2)				
350	(0 0 u 2)				
351	(1 0 u 2)				
352	(2 0 u 2)				
353	(3 0 u 2)				
354	(u 0 u 2)	2		-[+>+]<	-[+>+]<
355	(0 1 u 2)				
356	(1 1 u 2)				
357	(2 1 u 2)				
358	(3 1 u 2)				
359	(u 1 u 2)				
360	(0 2 u 2)				
361	(1 2 u 2)				
362	(2 2 u 2)				
363	(3 2 u 2)				
364	(u 2 u 2)	1	+[++]++	+[++]++	+[++]++
365	(0 3 u 2)				
366	(1 3 u 2)				
367	(2 3 u 2)				
368	(3 3 u 2)				
369	(u 3 u 2)				

370	(0 u u 2)				
371	(1 u u 2)				
372	(2 u u 2)				
373	(3 u u 2)				
374	(u u u 2)	1	+[]++	+[]++	+[]++
375	(0 0 0 3)	3			<[->+]<
376	(1 0 0 3)				
377	(2 0 0 3)	2		>+[+>]<	-[>+]>+
378	(3 0 0 3)				
379	(u 0 0 3)				
380	(0 1 0 3)	2		[>++]>++	[<++]>++
381	(1 1 0 3)	2		[>+]>++	[>+]>++
382	(2 1 0 3)	2		>[++>]<	+<[>+]<
383	(3 1 0 3)	2		[>-]>++	[>-]>++
384	(u 1 0 3)				
385	(0 2 0 3)				
386	(1 2 0 3)				
387	(2 2 0 3)	2		-[>-]>+	-[>-]>+
388	(3 2 0 3)				
389	(u 2 0 3)				
390	(0 3 0 3)				
391	(1 3 0 3)				
392	(2 3 0 3)	2		-[>++]>+	>[+<++]<
393	(3 3 0 3)	3			++>[-<]<
394	(u 3 0 3)				
395	(0 u 0 3)				
396	(1 u 0 3)				
397	(2 u 0 3)				
398	(3 u 0 3)				
399	(u u 0 3)				
400	(0 0 1 3)				
401	(1 0 1 3)				
402	(2 0 1 3)				
403	(3 0 1 3)	3			<[->+]>-
404	(u 0 1 3)				
405	(0 1 1 3)	2		+[++>-]<	+[++>-]<
406	(1 1 1 3)	3			<-[>+]<
407	(2 1 1 3)	2		++[>-]<	++[>-]<
408	(3 1 1 3)	3			>[->->]<
409	(u 1 1 3)				
410	(0 2 1 3)				
411	(1 2 1 3)				
412	(2 2 1 3)				
413	(3 2 1 3)	2		+[>+]<	+[>+]<
414	(u 2 1 3)				
415	(0 3 1 3)				

416	(1 3 1 3)				
417	(2 3 1 3)				
418	(3 3 1 3)	3			-<+[>-]<
419	(u 3 1 3)				
420	(0 u 1 3)				
421	(1 u 1 3)				
422	(2 u 1 3)				
423	(3 u 1 3)				
424	(u u 1 3)				
425	(0 0 2 3)	3		->-[+>]<	>[+>-]<
426	(1 0 2 3)				
427	(2 0 2 3)				
428	(3 0 2 3)				
429	(u 0 2 3)				
430	(0 1 2 3)	1	*	*	*
431	(1 1 2 3)	3		+>+[>-]<	<+[>]<
432	(2 1 2 3)	3		++[>++]<	<++[>]<
433	(3 1 2 3)	3		->-[>+]<	<-[>]<
434	(u 1 2 3)				
435	(0 2 2 3)	2		>+[>-]>+	-<+[+>]<
436	(1 2 2 3)				
437	(2 2 2 3)	3			+<[->]>-
438	(3 2 2 3)	2		+[+>++]<	+[+>++]<
439	(u 2 2 3)				
440	(0 3 2 3)	2		+[>++]>+	[+<+<+]<
441	(1 3 2 3)				
442	(2 3 2 3)				
443	(3 3 2 3)	3			[<+<+]<
444	(u 3 2 3)				
445	(0 u 2 3)				
446	(1 u 2 3)				
447	(2 u 2 3)				
448	(3 u 2 3)				
449	(u u 2 3)				
450	(0 0 3 3)	2		[+>++]>+	[+<++]>+
451	(1 0 3 3)	2		+[>-]>++	>[++<-]<
452	(2 0 3 3)				
453	(3 0 3 3)	3			->[-<]<
454	(u 0 3 3)				
455	(0 1 3 3)	2		-[++>+]<	-[++>+]<
456	(1 1 3 3)	3			>[+>+>]<
457	(2 1 3 3)	2		++[>+]<	++[>+]<
458	(3 1 3 3)	3			<+[>-]<
459	(u 1 3 3)				
460	(0 2 3 3)				
461	(1 2 3 3)	2		>-[>+]<	+<-[>]<

462	(2 2 3 3)				
463	(3 2 3 3)	3			+[<+<+]<
464	(u 2 3 3)				
465	(0 3 3 3)	3			>[-<]<
466	(1 3 3 3)	3			<+[<-<]<
467	(2 3 3 3)	3			<[+>]>++
468	(3 3 3 3)	2	[+]-	>-	>-
469	(u 3 3 3)				
470	(0 u 3 3)				
471	(1 u 3 3)				
472	(2 u 3 3)				
473	(3 u 3 3)				
474	(u u 3 3)				
475	(0 0 u 3)				
476	(1 0 u 3)				
477	(2 0 u 3)				
478	(3 0 u 3)				
479	(u 0 u 3)				
480	(0 1 u 3)				
481	(1 1 u 3)				
482	(2 1 u 3)				
483	(3 1 u 3)				
484	(u 1 u 3)	2		-[+>+]>+	-[+>+]>+
485	(0 2 u 3)				
486	(1 2 u 3)				
487	(2 2 u 3)				
488	(3 2 u 3)				
489	(u 2 u 3)				
490	(0 3 u 3)				
491	(1 3 u 3)				
492	(2 3 u 3)				
493	(3 3 u 3)				
494	(u 3 u 3)	1	+[++]-	+[++]-	+[++]-
495	(0 u u 3)				
496	(1 u u 3)				
497	(2 u u 3)				
498	(3 u u 3)				
499	(u u u 3)	1	+[]-	+[]-	+[]-
500	(0 0 0 u)				
501	(1 0 0 u)				
502	(2 0 0 u)				
503	(3 0 0 u)				
504	(u 0 0 u)	3			[-[>+]>]
505	(0 1 0 u)				
506	(1 1 0 u)				
507	(2 1 0 u)				

508	(3 1 0 u)				
509	(u 1 0 u)				
510	(0 2 0 u)				
511	(1 2 0 u)				
512	(2 2 0 u)				
513	(3 2 0 u)				
514	(u 2 0 u)				
515	(0 3 0 u)				
516	(1 3 0 u)				
517	(2 3 0 u)				
518	(3 3 0 u)				
519	(u 3 0 u)				
520	(0 u 0 u)	1	[++]	[++]	[++]
521	(1 u 0 u)				
522	(2 u 0 u)	2		++[+>+]<	++[+>+]<
523	(3 u 0 u)				
524	(u u 0 u)	1	++[]	++[]	++[]
525	(0 0 1 u)				
526	(1 0 1 u)				
527	(2 0 1 u)				
528	(3 0 1 u)				
529	(u 0 1 u)				
530	(0 1 1 u)				
531	(1 1 1 u)				
532	(2 1 1 u)				
533	(3 1 1 u)				
534	(u 1 1 u)				
535	(0 2 1 u)				
536	(1 2 1 u)				
537	(2 2 1 u)				
538	(3 2 1 u)				
539	(u 2 1 u)				
540	(0 3 1 u)				
541	(1 3 1 u)				
542	(2 3 1 u)				
543	(3 3 1 u)				
544	(u 3 1 u)				
545	(0 u 1 u)				
546	(1 u 1 u)	1	[++]+	[++]+	[++]+
547	(2 u 1 u)	2		[++>+>]<	[++<+>]<
548	(3 u 1 u)	2		[+>+]>-	[+>+]>-
549	(u u 1 u)	1	++[]+	++[]+	++[]+
550	(0 0 2 u)				
551	(1 0 2 u)				
552	(2 0 2 u)				
553	(3 0 2 u)				

554	(u 0 2 u)				
555	(0 1 2 u)				
556	(1 1 2 u)				
557	(2 1 2 u)				
558	(3 1 2 u)				
559	(u 1 2 u)				
560	(0 2 2 u)				
561	(1 2 2 u)				
562	(2 2 2 u)				
563	(3 2 2 u)				
564	(u 2 2 u)				
565	(0 3 2 u)				
566	(1 3 2 u)				
567	(2 3 2 u)				
568	(3 3 2 u)				
569	(u 3 2 u)				
570	(0 u 2 u)	2		[+>+]<	[+>+]<
571	(1 u 2 u)				
572	(2 u 2 u)	1	[++]++	[++]++	[++]++
573	(3 u 2 u)				
574	(u u 2 u)	1	++[]++	++[]++	++[]++
575	(0 0 3 u)				
576	(1 0 3 u)				
577	(2 0 3 u)				
578	(3 0 3 u)				
579	(u 0 3 u)				
580	(0 1 3 u)				
581	(1 1 3 u)				
582	(2 1 3 u)				
583	(3 1 3 u)				
584	(u 1 3 u)				
585	(0 2 3 u)				
586	(1 2 3 u)				
587	(2 2 3 u)				
588	(3 2 3 u)				
589	(u 2 3 u)				
590	(0 3 3 u)				
591	(1 3 3 u)				
592	(2 3 3 u)				
593	(3 3 3 u)				
594	(u 3 3 u)				
595	(0 u 3 u)				
596	(1 u 3 u)	2		[+>+]>+	[+>+]>+
597	(2 u 3 u)	2		[++>->]<	[++<->]<
598	(3 u 3 u)	1	[++]-	[++]-	[++]-
599	(u u 3 u)	1	++[]-	++[]-	++[]-

600	(0 0 u u)	2		[[->]->]	[[-<]->]
601	(1 0 u u)				
602	(2 0 u u)				
603	(3 0 u u)				
604	(u 0 u u)	1	-[]	-[]	-[]
605	(0 1 u u)				
606	(1 1 u u)				
607	(2 1 u u)				
608	(3 1 u u)				
609	(u 1 u u)	1	-[]+	-[]+	-[]+
610	(0 2 u u)				
611	(1 2 u u)				
612	(2 2 u u)				
613	(3 2 u u)				
614	(u 2 u u)	1	-[]++	-[]++	-[]++
615	(0 3 u u)				
616	(1 3 u u)				
617	(2 3 u u)				
618	(3 3 u u)				
619	(u 3 u u)	1	-[]-	-[]-	-[]-
620	(0 u u u)	1	[]	[]	[]
621	(1 u u u)	1	[]+	[]+	[]+
622	(2 u u u)	1	[]++	[]++	[]++
623	(3 u u u)	1	[]-	[]-	[]-
624	(u u u u)	2	[[]+]	>+[]	>+[]

C Construction of a Particular Function

The following output is obtained when one runs the `(demonstrate-build-alpha)` procedure given in the PROGRAM-GENERATORS.SCM file (see Appendix D.2):

```
Here follows a demonstration of the build-alpha rutine:

function f is (0 1 3 2 3 4 2 4)

function alpha should be (0 1 3 2 5 4 6 7)

alpha's implementation (in macro language) is: RRRRRRSHRRHRRRRSHRRRRH

alpha's implementation (in raw form) is:
++++++->->[+>]<->[-]>[-]<<++>[-]>[-]<<++++->->[+>]<->[-]>[-]<<++++>[-]>[-]<<

and the function actually implemented by alpha is (0 1 3 2 5 4 6 7)
```

TODO: Give a similar demo of the whole `build-f` procedure.

D Source Code

D.1 GENERIC.SCM

```
;*****generic functions******
; contains definitions not specific to this project

#lang scheme

(provide enum-nat
         mod
         enum-range
         enum-between
         filter
         repeat
         sort-add-string
         program<?
         string-dup
         dewhitespace
         set-subtract
         list->lambda
         integral
         index-of-in-list)

;enumerate natural numbers up to n: ( 0 .. n )
(define (enum-nat n)
  (enum-nat-helper n '()))
(define (enum-nat-helper n acc)
  (if (< n 0)
    acc
    (enum-nat-helper (- n 1) (cons n acc))))

;modulus (which also works for negative x)
(define (mod x n)
  (let ((y (remainder x n)))
    (if (< y 0)
      (mod (+ y n) n)
      y)))

;enumerate a range of natural numbers: ( n .. m )
(define (enum-range n m)
  (map (lambda (x) (+ n x)) (enum-nat m)))

(define (enum-between n m)
 (if (> n m)
   '()
   (enum-range n (- m n))))

;return those elements of lst for which pred? is true
; (maintains ordering of lst)
```

```
(define (filter pred? lst)
  (cond ((null? lst)
         '())
        ((pred? (car lst))
         (cons (car lst)
               (filter pred? (cdr lst))))
        (else
          (filter pred? (cdr lst)))))

;nest a function f in itself n times
(define (repeat f n)
  (if (= n 0)
    (lambda (x) x)
    (lambda (x) (f ((repeat f (- n 1)) x)))))

;add a string s to a list of strings sorted in acending order,
; maintaining this ordering.
(define (sort-add-string s los)
  (cond ((null? los)            ;if los is empty
         (list s))              ; then simply add s.
        ((program<? s (car los)) ;if s is smaller than smallest element of los
         (cons s los))          ; then add it at the beginning.
        (else                   ;otherwise, add s deeper in the list.
         (cons (car los) (sort-add-string s (cdr los))))))

;returns true if program a is shorter than program b
; (or comes earlier in a sorting)
(define (program<? a b)
  (cond ((< (string-length a) (string-length b)) #t)
        ((> (string-length a) (string-length b)) #f)
        (else (string<? a b))))


;repeat a string str times
(define (string-dup str times)
  (if (zero? times)
    ""
    (string-append str (string-dup str (- times 1)))))

;remove all whitespace from a string
(define (dewhitespace s)
  (cond ((zero? (string-length s))
         "")
        ((char-whitespace? (car (string->list s)))
         (dewhitespace (list->string (cdr (string->list s)))))
        (else
          (string-append
            (list->string (list (car (string->list s))))
                  (dewhitespace (list->string
                                  (cdr (string->list s))))))))
```

```
(define (set-subtract A B) ;A - B
  (cond ((null? A)
         '())
        ((member (car A) B)
         (set-subtract (cdr A) B))
        (else
          (cons (car A) (set-subtract (cdr A) B)))))


;(define (integrate-list l)
;  (integrate-list-helper l 0))
;
;(define (integrate-list-helper l acc)
;  (cond ((null? l) '())
;        (else (cons acc (integrate-list-helper (cdr l) (+ acc (car l)))))))

(define (integral f)
  (lambda (x)
    (apply + (map f (enum-nat (- x 1))))))

(define (list->lambda l)
  (lambda (xp)
    (let ((x (mod xp (length l))))
      (cond ;we do not need to check for null because of the mod
        ((zero? x)
         (car l))
        (else
         ((list->lambda (cdr l)) (- x 1)))))))



(define (index-of-in-list thing lst)
  (cond ((null? lst) (car '())) ;big fat error
        ((eq? thing (car lst))
         0)
        (else
          (+ 1 (index-of-in-list thing (cdr lst))))))
```

D.2 PROGRAM-GENERATORS.SCM

```
;**** program-generators.scm ****

#lang scheme
(provide hash->program
         Q-syntax?
         instr-base
         apply-macros
         build-alpha
         build-f
         theoretical-alpha
         build-alpha-helper ;delme
         build-mov
         )

(require "generic.scm")


;number of brainfsck commands
(define instr-base 7)

;table of brainfsck commands
(define (Q-cmd i)
  (cond ((= i 0) ">")
        ((= i 1) "<")
        ((= i 2) "+")
        ((= i 3) "-")
        ((= i 4) ".")
        ((= i 5) "[")
        ((= i 6) "]")))

;return the bf program corresponding to the given Goedel number (hash)
; (this is the p = #^-1(n) function)
(define (hash->program hash)
  (if (zero? hash)
    ""
    (string-append (Q-cmd (remainder hash instr-base))
                   (hash->program (quotient hash instr-base)))))

;an alternate approach to generating programs from hashes and filtering
; for syntactically correct programs is to generate them using a formal
; grammar. This program adds one more theorem to the given set, and can
; serve as a constructive Goedel numbering of theorems (if the pseudo-
; random function is sufficiently random).
;
;NOTE: the pseudorandom function should range between zero and at least
; an order-of-magnitude greater than the greatest Goedel number to be
; assigned. It should also be DETERMINISTIC if repeatable Goedel numberings
; are desired.
```

```
;
(define (index-list lst i)
  (if (zero? i)
    (car lst)
    (index-list (cdr lst) (- i 1))))
(define initial-program-set '("." ">" "<" "+" "-"))
(define (augmented-program-set program-set)
  (let* ((gn (+ 0 (string-length program-set)))
         ;choose two existing programs
         (a (index-list program-set (random gn)))
         ;increase occurance of basic commands
         (b (if (< (random 100) 80)
              (index-list program-set (random 5))
              (index-list program-set (random gn))))
         ; apply production rules
         (ab (string-append a b))
         (ba (string-append b a))
         (BaB (string-append "[" a "]")))
    ;add those new programs which are not already members
    (append program-set
            (filter (lambda (p-candidate)
                      (not (member p-candidate program-set)))
                    (if (< (random 100) 50)
                      (list ab ba BaB)
                      (list ab ba))))))

;check if program p is syntactically correct
; (that is, '[' and ']' symbols are matched)
(define (Q-syntax? p) (Q-syntax-helper p 0))
(define (Q-syntax-helper p n)
  (let  ((p-tail (if (< (string-length p) 2)
                   ""
                   (substring p 1)))
         (p-head (if (= (string-length p) 0)
                   ""
                   (substring p 0 1))))
  (cond ((< n 0) #f) ;too many close brackets
        ((equal? "" p-head) ;reached end of program
         (if (zero? n)
           #t ;gramatically correct program
           #f)) ;too many open brackets
        ((equal? "[" p-head)
         (Q-syntax-helper p-tail ;remove one char from program
                          (+ n 1))) ;n indicates depth of bracket-nesting
        ((equal? "]" p-head)
           (Q-syntax-helper p-tail (- n 1)))
        (else ;some other charicter - doesnt effect grammar
          (Q-syntax-helper p-tail n)))))
```

```
(define (apply-macros p)
  (if (zero? (string-length p))
    ""
    (string-append
      (let ((c (car (string->list p))))
        (cond
          ;((char=? c #\M) "<[>+]>")
          ((char=? c #\R) "+")
          ((char=? c #\S) "->->[+>]<-")
          ((char=? c #\G) "[->-<]>[>-]>")
          ((char=? c #\H) ">[-]>[-]<<") ;clean next two cells
          ;((char=? c #\D) "") ;THIS ONE TOO!
          ;((char=? c #\N) "[->-<]>")
          ;((char=? c #\Z) "")
          (else (list->string (list c)))))
      (apply-macros (list->string (cdr (string->list p)))))))

;fit-macros does the reverse of apply-macros
;(define (fit-macros p) ;TODO

;given two function implementations, this function returns the
; program implementing their composition
(define (composition->program func-impl-f func-impl-g)
  (string-append
    func-impl-g
    ;(apply-macros "H") ;clean up after first implementation
    "H"
    func-impl-f))

;takes a list of implementations, in PROGRAM ORDER (NOT COMPOSITION ORDER)
(define (multi-composition->program loi)
  (if (null? loi)
    ""
    (string-append
      (car loi)
      ;(apply-macros "H")
      "H"
      (multi-composition->program (cdr loi)))))

;returns a program that computes alpha[f]
; (using composition of R, G, and S)
(define (build-alpha f)
  (build-alpha-helper 0
                      (theoretical-alpha f)
                      (enum-nat (- (length f) 1)))) ;null function
(define (build-alpha-helper i af fp)
  (let* ((m (length fp)))
    (cond ((= m i) ;only satisfied if af==fp
           "") ;no more work to be done
```

```
          ((= ((list->lambda af) i)
              ((list->lambda fp) i))
           (build-alpha-helper (+ i 1) af fp)) ;the i^th entry is correct
          (else
            (let* ((desired-value ((list->lambda af) i))
                   (position-in-fp (index-of-in-list desired-value fp)))
              (composition->program
                (build-alpha-helper (+ i 1)
                                    af
                                    ;fix-up fp so that it represents what
                                    ; the program so far will produce
                                    (append
                                      (map (list->lambda fp)
                                           (enum-between 0
                                                         (- i 1)))
                                      (list desired-value)
                                      (map (list->lambda fp)
                                           (enum-between i
                                                         (- position-in-fp 1)))
                                      (map (list->lambda fp)
                                           (enum-between (+ position-in-fp 1)
                                                         (- m 1)))))
                (build-mov position-in-fp i m)))))))

;move j to i in a program, using base m
(define (build-mov j i m)
  (cond ((= i j) "") ;no movement to be done
        (else
          (string-append
            (build-mov (- j 1) i m)
            (make-string (mod (- m (- j 1)) m) #\R)
            "SH"
            (make-string (- j 1) #\R))))) ;R requires no housekeeping

;for reference, here is the alpha function as defined (not as constructed):
(define (theoretical-alpha f)
  (let* ((m (length f))
         (Z_m (enum-nat (- m 1)))
         (free (list->lambda (set-subtract Z_m f)))
         (dup (lambda (x) (dup-pred x 0 f))) ;eq for DUP given in footnote
         (dup-sum (integral dup)))
    (for/list ([x (enum-nat (- m 1))])
      (if (= 1 (dup x))
        (free (dup-sum x))
        ((list->lambda f) x)))))

(define (dup-pred x i f)
  (cond ((= x i)
         0)
        ((= (vector-ref (list->vector f) i)
```

```
            (vector-ref (list->vector f) x))
          1)
         (else
           (dup-pred x (+ i 1) f))))

;returns a program that computes f
; (using composition of R, G, and S)
(define (build-f f alpha) 0)
```

D.3 Q.SCM

```
;**** Q interpreter ****

#lang scheme
(provide interpreter
         pst-get-program
         pst-get-output
         halted?
         not-halted?
         pst-get-running-time
         Q-increment
         Q-decrement
         build-init-tuple
         order
         base)
(require "generic.scm")

(define order (make-parameter 2)) ;number of cells in memory ring
(define base (make-parameter 2)) ;number of states per cel

;interpreter executes one step from the given prog-state-tuple consisting of:
;
; prog-state-tuple =
; ( program
;     program-pointer
;     memory
;     memory-pointer
;     stack
;     output
;     halted?
;     run-time
;     checkpoints)
;
; and returns the resulting prog-state-tuple (with the program unchanged)
;
;Q-in/decrement implement the '+' and '-' Q operations.
(define (Q-increment x) (mod (+ x 1) (base)))
(define (Q-decrement x) (mod (- x 1) (base)))

;some short-hand for extracting data from prog-state-tuples:
(define (pst-get-program pst) (car pst))
(define (pst-get-output pst) (cadddr (cddr pst)))
(define (halted? pst) (cadddr (cdddr pst)))
(define not-halted? (lambda (pst) (not (halted? pst))))
(define (pst-get-running-time pst) (cadddr (cddddr pst)))

;actual interpreter implementation
(define (interpreter prog-state-tuple)
```

```
;extract the various data from the pst
(let*((p (car prog-state-tuple))
      (pp (cadr prog-state-tuple))
      (m (caddr prog-state-tuple))
      (mp (cadddr prog-state-tuple))
      (stack (cadddr (cdr prog-state-tuple)))
      (output (cadddr (cddr prog-state-tuple)))
      ;HALT STATUS caddd dddr
      (running-time (cadddr (cddddr prog-state-tuple)))
      (checkpoints (cadddr (cddddr (cdr prog-state-tuple))))
      (current-instruction (lambda () (substring p pp (+ pp 1))))
      (apply-function-to-memory-at-pointer
        (lambda (func) (begin (vector-set! m
                                           mp
                                           (func (vector-ref m mp)))
                              m)))
      (halting (>= (+ pp 1) (string-length p))))

  ;take action depending on the current command
(cond ((or (zero? (string-length p)) ;null program simply halts
           (halted? prog-state-tuple)) ;this program has halted
       (list p pp m mp stack output #t running-time checkpoints))

      ;move memory pointer one position forward
      ((equal? (current-instruction) ">")
       (list p (+ pp 1) m (mod (+ mp 1) (order))
             stack output halting (+ 1 running-time) checkpoints))

      ;move memory pointer one position backward
      ((equal? (current-instruction) "<")
       (list p (+ pp 1) m (mod (- mp 1) (order))
             stack output halting (+ 1 running-time) checkpoints))

      ;output value at memory pointer
      ((equal? (current-instruction) ".")
       (list p (+ pp 1) m mp stack (append output (list (vector-ref m mp)))
             halting (+ 1 running-time) checkpoints))

      ;increment memoron at memory pointer
      ((equal? (current-instruction) "+")
       (list p (+ pp 1) (apply-function-to-memory-at-pointer Q-increment)
             mp stack output halting (+ 1 running-time) checkpoints))

      ;decrement memoron at memory pointer
      ((equal? (current-instruction) "-")
       (list p (+ pp 1) (apply-function-to-memory-at-pointer Q-decrement)
             mp stack output halting (+ 1 running-time) checkpoints))

      ;decend
      ((equal? (current-instruction) "[")
```

```
         ;keep track of states to catch endless looping ASAP
         (let ((state-hash (make-state-hash (string-length p) pp mp m)))
           (if (member state-hash checkpoints)
               (list p pp m mp stack output #f running-time checkpoints)
               (list p (+ pp 1) m mp
                     (cons pp stack)
                     output halting (+ 1 running-time)
                     (cons state-hash checkpoints)))))

        ;conditional loop
        ((equal? (current-instruction) "]")
         (if (zero? (vector-ref m mp)) ;on zero
             (list p
                   (+ pp 1) ;continue to next instruction
                   m mp
                   (cdr stack) ;remove entry from stack
                   output halting (+ 1 running-time) checkpoints)
             (list p
                   (car stack) ;loop back to address on stack
                   m mp
                   (cdr stack) ;remove entry from stack (will be reloaded)
                   output #f (+ 1 running-time) checkpoints))))))

(define (make-state-hash plen pp mp m)
  (cond ((vector? m)
         (+ pp
          (* plen mp)
          (* plen (order)
             (make-state-hash plen 0 0 (vector->list m)))))
        ((and (= pp 0) (= mp 0))
         (cond ((null? m) 0)
               (else
                 (+ (car m)
                    (* (base)
                       (make-state-hash plen 0 0 (cdr m)))))))
        (else
          (display "HORRIBLE CRASH!\n"))))


;initialize a program-state-tuple with the given program p
(define (build-init-tuple p)
  (list p
        0     ;start at the beginning
        (make-vector (order)) ;spawn a blank memory vector
        0     ;start at "head" of memory
        '()    ;empty stack
        '()    ;empty output
        #f   ;not (yet?) halted
        0 ;not yet run at all
        '())) ;no checkpoints
```

D.4 META-PROGRAMS.SCM

```
;**** metaprograms.scm ****
; allows high-level manipulation of Q programs

#lang scheme

(provide buisy-beaver
         big-omega
         explore
         output-equivalence
         find-running-time
         load-run-halt
         which-function
         explore-functions
         hash->function
         function->hash)

(require "generic.scm"
         "program-generators.scm"
         "Q.scm")


;generate a table of the buisy beaver function up to some program length n,
; with maximum cycles c:
(define (buisy-beaver n c)
    (buisy-beaver-helper (filter halted? (explore n c))
                         (make-vector (+ 1 n) 0)))
(define (buisy-beaver-helper psts records)
  (cond ((null? psts) records) ;no records can be broken if there are no psts
        ;otherwise we need to see if the first pst breaks a record
        (else (let* ((n (string-length (pst-get-program (car psts))))
                     (prev-record (vector-ref records n))
                     (this-length (length (pst-get-output (car psts))))
                     (new-records (if (< prev-record this-length)
                                      ;update new record if it is broken
                                      (begin
                                        (vector-set! records n this-length)
                                        records)
                                      records))) ;otherwise just use old records
          (buisy-beaver-helper (cdr psts) new-records)))))




;compute the halting probability by Chaitin's algorithm
;
;lim as k->inf of big-omega(k) is omega
; however, this converges slower than *any* recursive function,
```

```
;  as shown by Chaitin.
;note: we dont count programs of length 0, so the first k is 1
(define (big-omega k) (big-omega-helper 1 k 0 '()))
(define (big-omega-helper k kmax acc psts)
  (cond ((> k kmax)
         acc)
        (else
          (let* ((little-o (omega-partial-sum k psts))
                 (partial-sum (car little-o))
                 (new-psts (cadr little-o)))
            (begin
              (display "big-omega(") (display k) (display ")=")
              (display (+ acc partial-sum)) (newline)
              (big-omega-helper (+ k 1)
                                kmax
                                (+ acc partial-sum)
                                new-psts))))))
; The output of omega-partial-sum is actually the kth partial sum and the
;  as-yet un-halted program-states of length k bits advanced k cycles.
;
; This sum denotes the quantity contributed to omega by the programs of length
;  less than k bits which terminate in exactly k cylces, plus that quantity
;  contributed by programs of length exactly k bits which terminate in less
;  than k cycles.
(define (omega-partial-sum k psts)
  (let* ((programs-of-length-k
           (map build-init-tuple  ;construct new programs
                (filter Q-syntax?
                        (map hash->program (enum-range (expt instr-base (- k 1))
                                                       (- (expt instr-base k) 1))))))
         ;advance programs of length < k one step
         (new-psts (append (map interpreter
                                (filter not-halted? psts))
                           (map (repeat interpreter k)
                                programs-of-length-k))))
    ;advance new programs k steps
    (list (apply + (map
                     (lambda (x)
                       (if (halted? x)
                         (* (- 1 instr-base)
                            (expt instr-base
                                  (* -2
                                     (string-length (pst-get-program x)))))
                         0))
                     new-psts)) ;partial sum
          new-psts)))
```

```
;explore runs all programs of length up-to n, and displays output of programs
; after c cycles.
(define (explore n c)
  (begin
    (define l 0)
    (display "enumerating hashes...\n")
    (set! l (enum-nat (- (expt instr-base n) 1))) ;initial programs
    (display "de-hasing...\n")
    (set! l (map hash->program l))
    (display "applying syntax filter...\n")
    (set! l (filter Q-syntax? l))
    (display "constructing initial program-state-tuples...\n")
    (set! l (map build-init-tuple l))
    (display "executing programs...\n")
    (display "INIT--10%-----20%-----30%-----40%-----50%-----60%-----70%-----80%-----90%--DONE\n")
    (for ([i (cdr (enum-nat c))]) ;i indicates cycle number
       ;show progress every 10%:
       (if (zero? (remainder i (quotient c 80)))
         (begin
           (display "*")
           (flush-output))
         0)
       (set! l (map interpreter l)))
    (newline)
    l)) ;return the processed psts




;build an equivalence relation amoung the psts given
;
;The return value gives a list of output equivalence classes, that is programs
;  which return the same output. The format for an equivalence class is:
;      ((output) (list of programs in acending complexity order))
;
;In the case of programs which have not halted after c cylces, they are placed
; in a seperate output equivalence class ; with other yet-unhalted programs
; which have given the same output. (Note: these unhalted classes are just
; guesses two unhalted programs in the same class may actually develop differ-
; ent output, one or both may halt - likewise two programs in different classes
; may actually be output-equivalent on a long enough (even infinite) timescale)
;
; TODO: Actually implement the unhalted output classification.
;
(define (output-equivalence psts)
  (output-equivalence-append psts '()))
;processes the given program-state-tuples and addes them to the given classes
```

```
(define (output-equivalence-append psts classes)
  (cond ((null? psts) classes)
;NOTE: unhalted output classification UNIMPLEMENTED
        ((halted? (car psts))
         (output-equivalence-append
           (cdr psts)
           (augment-classes (car psts) classes)))
        (else
          (output-equivalence-append
            (cdr psts)
            classes))))
(define (augment-classes pst classes)
  (cond ((null? classes) ;no match -> add new entry
         ;initial entry contains output and first program
         (list (list (pst-get-output pst) (list (pst-get-program pst)))))
        ((equal? (caar classes) (pst-get-output pst)) ;found match
         (cons (list (caar classes)
                     (sort-add-string (pst-get-program pst) (cadar classes)))
               ;no (more) recursion nessisary, as we have just made the entry
               (cdr classes)))
        (else ;continue searching
          (cons (car classes) (augment-classes pst (cdr classes))))))
```

```
(define (find-running-time p)
  (find-running-time-of-pst (build-init-tuple p)))
(define (find-running-time-of-pst pst)
  (if (halted? pst)
    (cadddr (cddddr pst)) ;extract running-time field
    (find-running-time-of-pst (interpreter pst))))

(define (load-run-halt program)
  (pst-get-output (load-run-halt-helper (build-init-tuple program) 1000)))
(define (load-run-halt-helper pst limit)
  (let ((new-pst (interpreter pst)))
    (if (or (halted? new-pst)
            (>= 0 limit)
            (= (cadr pst) (cadr new-pst))) ;loop-detect
      new-pst
      (load-run-halt-helper new-pst (- limit 1)))))


;which-function, given a program string, will return the function which that
; program implements. The program is given a number in the range 0..m in the
; first memory cell and, after halting, the "output" is the value of that
; first memory cell. The so-called output string is ignored entirely.
(define (which-function p)
  (for/list ([arguement (enum-range 0 (- (base) 1))])
```

```
    (let* ((pre-pst (list
                     p
                     0     ;start at the beginning
                     (list->vector
                       (cons arguement
                             (vector->list
                               (make-vector (- (order) 1))))) ;memory has
                     0     ;start at "head" of memory         ; arguement as
                     '()    ;empty stack                      ; first cell
                     '()    ;empty output
                     #f   ;not (yet?) halted
                     0 ;not yet run at all
                     '())) ;no checkpoints
           (c (* (order)
              (expt (base) (order))
              (string-length p))) ;max halting time
           (post-pst (load-run-halt-helper pre-pst c))
           (ret-val (if (halted? post-pst)
                     (vector-ref (caddr post-pst)
                                 (cadddr post-pst)) ;grab cell at mp
                     'u))) ;indicate that the program did not halt
      ret-val)))

(define (hash->function hash m)
  (let ((f (hash->function-helper hash m)))
    (append f (vector->list (make-vector (- m (length f)) 0)))))
(define (hash->function-helper hash m)
  (cond ((zero? hash) '())
        (else
          (cons (if (= m (remainder hash (+ 1 m)))
                    'u
                    (remainder hash (+ 1 m)))
                (hash->function-helper (quotient hash (+ m 1)) m)))))

(define (function->hash f m) (function->hash-helper f m 0))
(define (function->hash-helper f m i) ;i should not be initially specified
  (cond ((= (length f) i)
         0)
        (else
          (+ (* (expt (+ m 1) i)
                (let ((element (vector-ref (list->vector f) i)))
                  (if (eq? 'u element)
                      m
                      element)))
             (function->hash-helper f m (+ i 1))))))


;explore-functions runs all programs of length up-to n,
; and constructs a vector holding the elegant implementations of the m^m
; after c cycles.
```

```
(define (explore-functions max-prog-len max-order)
  (let ((v (make-vector (expt (+ (base) 1) (base)) '())))
   (begin
    (define l 0)
    (display "enumerating hashes...\n")
    (set! l (enum-nat (- (expt instr-base max-prog-len) 1))) ;initial programs
    (display "de-hasing...\n")
    (set! l (map hash->program l))
    (display "applying syntax filter...\n")
    (set! l (filter Q-syntax? l))
    (newline)
    (for ([current-order (enum-range 1 (- max-order 1))])
      (order current-order)
      (set! v (explore-functions-augment l max-prog-len v)))
    v)))

(define (explore-functions-augment l max-prog-len v)
  (let ((c (* (order)
              (expt (base) (order))
              (+ max-prog-len 1)))
        (i 0)
        (length-l (- (expt instr-base max-prog-len) 1)))
        ;c is the maximum number of cycles to run programs
  (begin
    ;v will contain the discovered implementations

    (display "executing programs with n=") (display (order)) (newline)
    (display "INIT--10%-----20%-----30%-----40%-----50%-----60%-----70%-----80%-----90%--DONE\n")
    (for ([p l])
     (set! i (+ 1 i))
     (if (zero? (remainder i (quotient length-l 80)))
         (begin
           (display "*")
           (flush-output))
         0)
      (let* ((f (which-function p))
             (hash (function->hash f (base)))
             (others (vector-ref v hash))
             (elegant? (not (member (order) (map cadr others)))))
        (if elegant?
          (vector-set! v hash (cons (list p (order)) others)) 0)))
    v)))
```

D.5 EXPERIMENTS.SCM

```
;**** experiments.scm ***

#lang scheme

(provide print-output-classes
         print-buisy-beaver
         tabulate-output-classes
         print-function-table
         demonstrate-build-alpha)

(require "generic.scm"
         "program-generators.scm"
         "Q.scm"
         "meta-programs.scm")

(define (print-output-classes)
  (let ((psts 0) (classes 0) (count 0))
  (begin
    (display "Determining output equivalence classes...") (newline)
    (set! psts (explore 6 200))
    (set! classes (output-equivalence psts))
    (display "DONE!\nOf the ") (display (length psts))
    (display " grammatical programs analized, ")
    (display (length classes))
    (display " distinct output classes were found:") (newline)
    (set! count 0)
    (map (lambda (class)
           (begin
             (set! count ( + 1 count ))
             (display count) (display ". ") (display "Class ")
             (display (car class)) (display " has ")
             (display (length (cadr class)))
             (if (= 1 (length (cadr class)))
               (display " member: ")
               (display " members, with elegant program: "))
             (display (caadr class))
             (display " (HT=") (display (find-running-time (caadr class)))
                    (if (not (zero? (length (car class))))
                      (begin (display "c, NCR=")
                             (display (exact->inexact
                                       (/ (string-length (caadr class))
                                          (length (car class))))))
                      (display "c"))
                    (display ")")
             (newline)))
         classes)
    (newline))))
```

```
(define (tabulate-output-classes)
  (let ((psts 0) (classes 0) (count 0))
  (begin
    (display "Determining output equivalence classes...") (newline)
    (set! psts (explore 6 200))
    (set! classes (output-equivalence psts))
    (display "DONE!\nOf the ") (display (length psts))
    (display " grammatical programs analized, ")
    (display (length classes))
    (display " distinct output classes were found:") (newline)
    (set! count 0)
    (map (lambda (class)
           (begin
             (set! count ( + 1 count ))
             (display count) (display "\t")
             (display (car class)) (display "\t")
             (display (length (cadr class)))
             (display "\t")
             (display (caadr class))
             (display "\t")
             (display (find-running-time (caadr class)))
                    (if (not (zero? (length (car class))))
                      (begin (display "\t")
                             (display (exact->inexact
                                       (/ (string-length (caadr class))
                                          (length (car class))))))
                      (display "\t"))
             (newline)))
         classes)
    (newline))))



(define (print-buisy-beaver)
  (newline)
  (for ([i (cdr (enum-nat 10))])
    (order i)
    (display "USING ORDER = ") (display order) (newline)
    (display (buisy-beaver 6 1000)))
  (newline))


;assumes base is already chosen
(define (print-function-table max-prog-len max-order)
 (let ((v (explore-functions max-prog-len max-order)))
  (begin
    (newline)
    (display "FUNCTION TABLE FOR BASE ")
    (display (base))
```

```
    (display ", ORDER UP TO ")
    (display max-order)
    (display ", AND PROG-LEN UP TO ")
    (display max-prog-len)
    (newline)
    (display "id\tfunc\tN_H\tprograms of order n=1..") (display max-order)
    (newline)

    (for ([hash (enum-nat (- (vector-length v) 1))])
      (display hash) (display "\t")
      (display (hash->function hash (base))) (display "\t")
      (display (N_H (vector-ref v hash))) (display "\t")
      (for ([n (enum-range 1 max-order)])
        (let ((ps (vector-ref v hash)))
          (cond ((not (member n (map cadr ps)))
                 (display (make-string (+ 0 max-prog-len) #\ ))
                 (display "\t"))
                (else
                  (let ((p (caar (filter (lambda (x)
                                           (= (cadr x) n))
                                         ps))))
                    (begin
                      (if (zero? (string-length p))
                        (begin (set! p "*") (display "*"))
                        (display p))
                      (display (make-string (+ 0 max-prog-len
                                               (- (string-length p)))
                                            #\ ))
                      (display "\t")))))))))
      (newline)))))

(define (N_H ps)
  (if (null? ps) ""
  (let ((elegant-length (apply min (map (lambda (x) (string-length (car x)))
                                        ps))))
    (apply min (map cadr
                    (filter (lambda (x) (= (string-length (car x))
                                           elegant-length))
                            ps))))))


(define (demonstrate-build-alpha)
  (begin
    (base 8)
    (order 3)
    (let* ((f '(0 1 3 2 3 4 2 4))
           (alpha-macro (build-alpha f))
           (alpha-raw (apply-macros alpha-macro)))
    (display "Here follows a demonstration of the build-alpha rutine:")
    (newline) (newline)
```

```
(display "function f is ")
(display f) (newline) (newline)
(display "function alpha should be ")
(display (theoretical-alpha f)) (newline)(newline)
(display "alpha's implementation (in macro language) is: ")
(display alpha-macro) (newline)(newline)
(display "alpha's implementation (in raw form) is: ")
(display alpha-raw) (newline)(newline)
(display "and the function actually implemented by alpha is ")
(display (which-function alpha-raw)) (newline)(newline)

)))
```

D.6 MAIN.SCM

```
;***********************
;*  Some Experiments   *
;* on a Finite Machine *
;*                     *
;* compute the halting *
;* probability for the *
;* brainfsck computer  *
;* by Chaitin's method *
;* and explore some    *
;* properties of a fi- *
;* nite machine via    *
;* experimentation.    *
;*                     *
;*    11 July 2008     *
;*    Chris  Merck     *
;*  Stevens Institute  *
;*    of Technology    *
;*                     *
;***********************


;  List of Ideas to Implement / Explore:
;    * COMPUTE THE HALTING PROBABILITY (for several interpreter variations
;        and/or Goedel numberings)
;    * EXPLORE THE OUTPUT EQUIVALENCE RELATION - to what extent can the equiva-
;        lence be reached topographically? For example a program P = A[B]C with
;        B not containing "." returns the same value (if any) as P' = A[-]C.
;    * DEMONSTRATE EFFECTIVENESS OF BOTH GOEDEL NUMBERINGS
;    * FIND THE DENSITY OF "interesting" PROGRAMS

#lang scheme

;import project's software
(require "generic.scm")
(require "program-generators.scm")
(require "Q.scm")
(require "meta-programs.scm")
(require "experiments.scm")



(begin

  ;USER: envoke experiments here

  (exit))
```