

A Relatively Small Turing Machine Whose Behavior Is Independent of Set Theory

Adam Yedidia

MIT

adamy@mit.edu

Scott Aaronson

MIT

aaronson@csail.mit.edu

May 9, 2016

Abstract

Since the definition of the Busy Beaver function by Radó in 1962, an interesting open question has been what the smallest value of n for which $BB(n)$ is independent of ZFC set theory. Is this n approximately 10, or closer to 1,000,000, or is it even larger? In this paper, we show that it is at most 7,910 by presenting an explicit description of a 7,910-state Turing machine Z with 1 tape and a 2-symbol alphabet that cannot be proved to run forever in ZFC (even though it presumably does), assuming ZFC is consistent. The machine is based on work of Harvey Friedman on independent statements involving order-invariant graphs. In doing so, we give the first known upper bound on the highest provable Busy Beaver number in ZFC. To create Z , we develop and use a higher-level language, Laconic, which is much more convenient than direct state manipulation. We also use Laconic to design two Turing machines, G and R , that halt if and only if there are counterexamples to Goldbach’s conjecture and Riemann’s hypothesis, respectively.

1 Introduction

1.1 Background and Motivation

Zermelo-Fraenkel set theory with the axiom of choice, more commonly known as ZFC, is an axiomatic system invented in the twentieth which has since been used as the foundation of most of modern mathematics. It encodes arithmetic by describing natural numbers as increasing sets of sets.

Like any axiomatic system capable of encoding arithmetic, ZFC is constrained by Gödel’s two incompleteness theorems. The first incompleteness theorem states that if ZFC is *consistent* (it never proves both a statement and its opposite), then ZFC cannot also be *complete* (able to prove every true statement). The second incompleteness theorem states that if ZFC is consistent, then ZFC cannot prove its own consistency. Because we have built modern mathematics on top of ZFC, we can reasonably be said to have assumed ZFC’s consistency. This means that we must also believe that ZFC cannot prove its own consistency. This fact carries with it certain surprising conclusions.

In particular, consider a Turing machine Z that enumerates, one after the other, each of the provable statements in ZFC. To describe how such a machine might be constructed, Z could iterate over the axioms and inference rules of ZFC, applying each in every possible way to each conclusion

or pair of conclusions that had been reached so far. We might ask Z to halt if it ever reaches a contradiction; in other words, Z will halt if and only if it finds a proof of $0 = 1$. Because this machine will enumerate *every* provable statement in ZFC, it will run forever if and only if ZFC is consistent.

It follows that Z is a Turing machine for which the question of its behavior (whether or not it halts when run indefinitely) is equivalent to the consistency of ZFC.¹ Therefore, just as ZFC cannot prove its own consistency (assuming ZFC is consistent), ZFC also cannot prove that Z will run forever. In other words, the statement, “ Z will run forever” is *independent of* ZFC.

This is interesting because, while the undecidability of the halting problem tells us that there cannot exist an algorithmic method for determining whether an *arbitrary* Turing machine loops or halts, Z is an example of a *specific* Turing machine whose behavior cannot be proven one way or the other using the foundation of modern mathematics. Mathematicians and computer scientists think of themselves as being able to determine how a given algorithm will behave if given enough time to stare at it; despite this intuition, Z is a machine whose behavior we can never prove without assuming axioms more powerful than those generally assumed in modern mathematics.

1.2 Turing Machines

There are many slightly different definitions of Turing machines. For example, some definitions allow the machine to have multiple tapes; others only allow it to have one; some allow an arbitrarily large alphabet, while others allow only two symbols, and so on. In most research regarding Turing machines, mathematicians don’t concern themselves with which of these models to use, because any one can simulate the others (usually efficiently). However, because this work is concerned with upper-bounding the exact number of states required to perform certain tasks, it’s important to define the model precisely. The model we choose here is traditional for the Busy Beaver function.

Formally, a k -state Turing machine is a 7-tuple $M = (Q, \Gamma, a, \Sigma, \delta, q_0, F)$, where:

Q is the set of k states $\{q_0, q_1, \dots, q_{k-2}, q_{k-1}\}$

$\Gamma = \{a, b\}$ is the set of *tape alphabet symbols*

a is the *blank symbol*

Σ is the set of *input symbols*

$\delta = Q \times \Gamma \rightarrow (Q \cup F) \times \Gamma \times \{L, R\}$ is the *transition function*

q_0 is the *start state*

$F = \{\text{HALT}, \text{ERROR}\}$ is the set of *halting states*.

A Turing machine’s *states* make up the Turing machine’s easily-accessible, finite memory. The Turing machine’s state is initialized to q_0 .

The *tape alphabet symbols* correspond to the symbols that can be written on the Turing machine’s infinite tape.

In this work, all Turing machines are run on the all- a input.

The *transition function* encodes the Turing machine’s behavior. It takes two inputs: the current state of the Turing machine (an element of $Q \cup F$) and the symbol read off the tape (an element of

¹While we will talk about ZFC throughout this paper, rather than simple ZF set theory, this is simply a convention. For our purposes, the Axiom of Choice is irrelevant: the consistency of ZFC is equivalent to the consistency of simple ZF set theory, [14] and ZFC and ZF prove exactly the same arithmetical statements (which include, among other things, statements about whether Turing machines halt). [23]

Γ). It outputs three instructions: what state to enter (an element of $Q \cup F$), what symbol to write onto the tape (an element of Γ) and what direction to move the head in (an element of $\{L, R\}$). A transition function specifies the entire behavior of the Turing machine in all cases.

The *start state* is the state that the Turing machine is in at initialization.

A *halting transition* is a transition to a halting state, which causes the Turing machine to halt. While having three possible halting transitions is not necessary for our purposes, being able to differentiate between different types of halting (HALT and ERROR) is useful for testing.

1.3 The Busy Beaver Function

Consider the set of all Turing machines with k states, for some positive integer k . We call a Turing machine B a *k-state Busy Beaver* if when run on the empty tape as input, B halts, and also runs for at least as many steps before halting as all other halting k -state Turing machines. [22]

In other words, a Busy Beaver is a Turing machine that runs for at least as long as all other halting Turing machines with the same number of states. Another common definition for a Busy Beaver is a Turing machine that writes as many 1's on the tape as possible; because the number of 1's written is a somewhat arbitrary measure, it is not used in this work.

The *Busy Beaver function*, written $BB(k)$, equals the number of steps it takes for a k -state Busy Beaver to halt. The Busy Beaver function has many striking properties. To begin with, it is not *computable*; in other words, there does not exist an algorithm that takes k as input and returns $BB(k)$, for arbitrary values of k . This follows directly from the undecidability of the halting problem. Suppose an algorithm existed to compute the Busy Beaver function; then given a k -state Turing machine M as input, we could compute $BB(k)$ and run M for $BB(k)$ steps. If, after $BB(k)$ steps, M had not yet halted, we could safely conclude that M would never halt. Thus, we could solve the halting problem, which we know is impossible.

By the same argument, $BB(k)$ must grow faster than any computable function. (To check this, assume that some computable function $f(k)$ grows faster than $BB(k)$, and substitute $f(k)$ for $BB(k)$ in the rest of the proof.) In particular, the Busy Beaver grows even faster than (for instance) the Ackermann function, a well-known fast-growing function.

Because finding the value of $BB(k)$ for a given k requires so much work (one must fully explore the behavior of all k -state Turing machines), few explicit values of the Busy Beaver function are known. The known values are [4, 16]:

$$BB(1) = 1$$

$$BB(2) = 6$$

$$BB(3) = 21$$

$$BB(4) = 107$$

For $BB(5)$, $BB(6)$, and $BB(7)$ only lower bounds are known: $BB(5) \geq 47,176,870$, $BB(6) > 7.4 \times 10^{36,534}$, and $BB(7) > 10^{10^{10^{10^7}}}$. Additionally, $BB(22)$ is known to be larger than Graham's Number (a famous huge number from Ramsey theory, obtained by iterating the Ackermann function 64 times) [19][8][9]. Researchers have worked on pinning down the value of $BB(5)$ exactly, and some consider it to be possibly within reach.

Another way to discuss the Busy Beaver sequence is to say that modern mathematics has established a *lower bound* of 4 on the highest provable Busy Beaver value. In this paper, we prove

the first known *upper bound* on the highest provable Busy Beaver value in ZFC; that is, we give a value of k , namely 7,910, such that the value of $BB(k)$ cannot be proven in ZFC.

Intuitively, one might expect that while no algorithm may exist to compute $BB(k)$ for *all* values of k , we could find the value of $BB(k)$ for any *specific* k using a procedure similar to the one we used to find the value of $BB(k)$ for $k \leq 4$. The reason this is not so is closely tied to the existence of a machine like the Gödelian machine Z , as described in Section 1.1. Suppose that Z has k states. Because Z 's behavior (whether it halts or loops) cannot be proven in ZFC, it follows that the value of $BB(k)$ also can't be proven in ZFC; if it could, then a proof would exist of Z 's behavior in ZFC. Such a proof would consist of a *computation history* for Z , which is an explicit step-by-step description of Z 's behavior for a certain number of steps. If Z halts, then a computation history leading up to Z 's halting would be the entire proof; if Z loops, then a computation history that takes $BB(k)$ steps, combined with a proof of the value of $BB(k)$, would constitute a proof that Z will run forever.

In this paper we construct a machine like Z , for which a proof that Z runs forever would imply that ZFC was consistent. In doing so, we give an explicit upper bound on the highest Busy Beaver value provable in ZFC assuming the consistency of a slightly stronger set theory. Our machine, which we shall refer to as Z hereafter, contains 7,910 states. Therefore, we will never be able to prove the value of $BB(7,910)$ without assuming more powerful axioms than those of ZFC. This upper bound is presumably very far from tight, but it is a first step.

Even to achieve a state count of 7,910, we will need three nontrivial ideas: Harvey Friedman's order-theoretic statements, *on-tape processing*, and *introspective encoding*. Without all three ideas, we found that the state count would be in the tens of thousands, hundreds of thousands, or even millions. We briefly introduce these ideas in the following subsection, and explore them in much greater detail in Section 8. The implementation of these ideas constitutes this paper's main technical contribution.

1.4 Parsimony

In most algorithmic study, efficiency is the primary concern. In designing Z , however, parsimony is the only thing that matters. One historical analogue is the practice of "code-golfing": a recreational pursuit adopted by some programmers in which the goal is to produce a piece of code in a given programming language, using as few characters as possible. Many examples of code-golfing can be found at [26]. The goal of designing a Turing machine with as few states as possible to accomplish a certain task, without concern for the machine's efficiency or space usage, can be thought of as code-golfing with a particularly low-level programming language.

Part of the charm of Turing machines is that they give us a "standard reference point" for measuring complexity, unencumbered by the details of more sophisticated programming languages. Also, with Turing machines, there can be no suspicion that we engineered a programming formalism just for the purpose of code-golfing, or for making the concepts we want artificially simple to describe. This is why we prefer Turing machines as a tool for measuring complexity; not because they are particularly special, but simply because they are so primitive that their specifics will interfere minimally with what we mean by an algorithm being "complicated."

In this paper, we use three ideas for generating parsimonious Turing machines: Harvey Friedman's mathematical statements, *on-tape processing*, and *introspective* Turing machines. The last of these ideas was proposed, under a different name and with some variations, by Ben-Amram and Petersen in 2002 [3]. These three ideas are explained in more detail in Subsections 3.1, 8.1, and 8.3,

respectively, but we summarize them very briefly here.

The first idea is simply to use the research done by Friedman into finding simple-to-express statements that are equivalent to the consistency of various axiomatic systems. In particular, we use a statement discovered by Friedman to be equivalent to the consistency of a set theory stronger than ZFC (and whose consistency, therefore, would imply the consistency of ZFC).² [10]

The second idea, on-tape processing, is a way to encode high-level commands into a Turing machine parsimoniously. Instead of converting commands to groups of states directly, which incurs a multiplicative overhead based on how large these groups need to be, on-tape processing begins by writing the commands onto the tape, using as efficient an encoding as possible. Then, once the commands are on the tape, the commands are processed by a single group of states that understands how to interpret them.

The third idea, introspective Turing machines, is a way to write long strings onto the tape using as few states as possible. The idea is to encode information one of each state’s transitions, instead of encoding information in each state’s write field. This is advantageous because there are many choices for which state to point a transition to, but only two choices for which bit to write. Therefore, more information can be encoded in each state using this method.

1.5 Implementation Overview

To generate descriptions of Turing machines with nice mathematical properties entirely by hand is a daunting task. Rather than approach the problem directly, we created tools for generating parsimonious Turing machines while presenting an interface that is comfortably familiar to most programmers (and to us!).

We created two tools. At the top level is the Laconic programming language, whose syntax and capabilities are similar to those of most programming languages, such as Java or Python. Beneath it we created a lower-level language called Turing Machine Descriptor (TMD). TMD is quite unlike most programming languages, and is better thought of as a convenient way to describe a multi-tape, 3-symbol Turing machine plus a function stack. The style of multi-tape Turing machine used in TMD is the commonly used “one-tape-at-a-time” abstraction: only one tape at a time can be interacted with, for reading, writing, and moving the head. Laconic compiles down to a TMD program, and TMD compiles down to a description of a single-tape, 2-symbol Turing machine. This process is illustrated in Figure 1.

We recommend that programmers hoping to use our tools to generate their own encodings of mathematical statements or algorithms as Turing machines use Laconic. Laconic’s interface is perfect for somebody hoping to write in a “traditional” language. On the other hand, if the programmer wishes to improve upon Laconic’s compilation process, writing code directly in TMD is likely to be the better option.

2 Related Work

Gregory Chaitin raised the problem of proving a version of our result in his book *The Limits of Mathematics*. [7] He wrote:

²It is unclear whether using Friedman’s statements in fact does decrease the state usage of the Turing machines created, or whether a direct approach would perform better. [1] However, we find it much easier to reason about programs encoding graph-theoretic statements than programs that enumerate proofs using first-order logic!

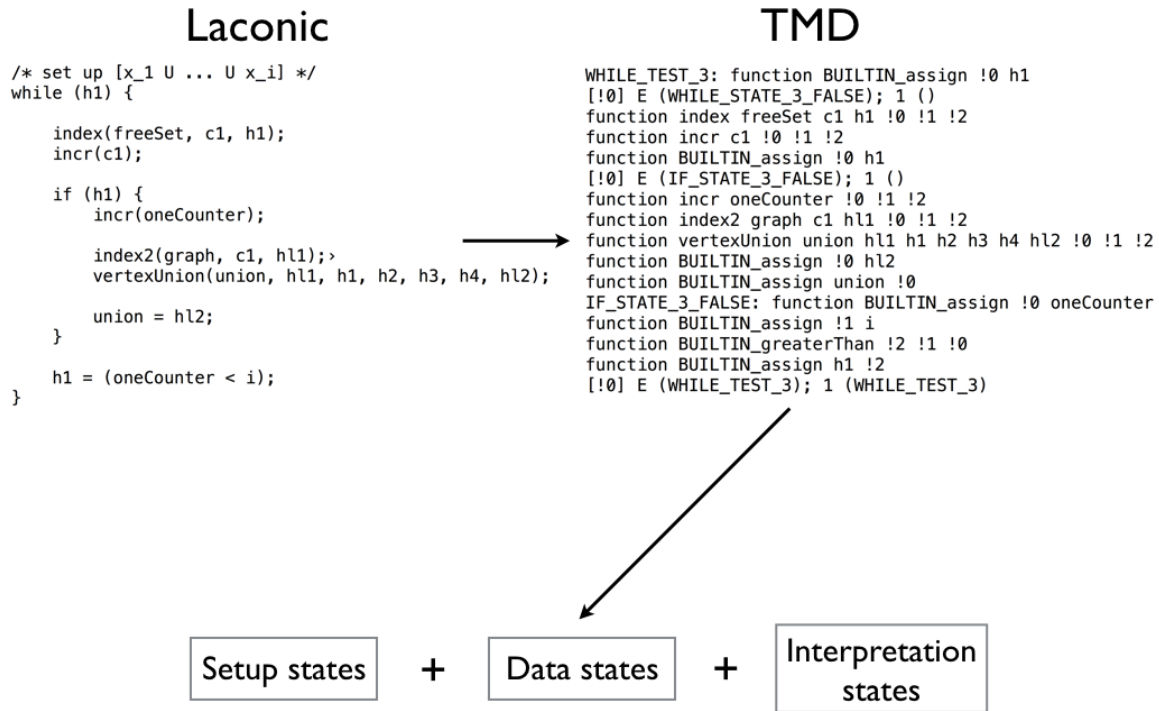


Figure 1: A visual overview of the compilation process.

I would like to have somebody program out Zermelo-Fraenkel set theory in my version of LISP, which is pretty close to normal LISP as far as this task is concerned, just to see how many bits of complexity mathematicians normally assume . . . If you programmed ZF, you'd get a really sharp incompleteness result. It wouldn't say that you can get at most $H(ZF) + 15328$ bits of [Chaitin's halting probability] Ω , it would say, perhaps, at most 96000 bits! We'd have a much more definite incompleteness theorem.

We did not program ZF set theory in LISP, but we programmed it in an even simpler language—thereby answering Chaitin's call for an explicit number of bits to attach to the complexity of ZF set theory. (As many as required to fully describe our Turing machine—or more precisely, 157,819.)

This paper is not the first to attempt to quantify the complexity of arithmetical statements. Calude and Calude [6] define a register machine of their own design, and provide quantifications of the complexity of Legendre's conjecture, Fermat's last theorem, Goldbach's conjecture, Dyson's conjecture, the Riemann hypothesis, and the four color theorem.³ In addition, Koza [15] and Pargellis [21] each invent instruction sets that are particularly well-suited to representing self-reproducing programs simply, and show that starting from a “primordial soup” of such instructions distributed about a large memory, along with an increasing number of program threads, a rich ecosystem of increasingly efficient self-reproducing programs start to dominate the “landscape.”

Also similar to the work of this paper is the famous search for a universal Turing machine. A *universal* Turing machine is a Turing machine that can simulate another Turing machine, when provided a description of the other machine on its input tape. The smallest universal Turing machine is a 2-state, 3-symbol Turing machine, found by Smith [24].

This paper differs from the previous work in two ways: firstly, it is the first to give explicit, relatively small machines whose behavior is provably independent of the standard axioms of modern mathematics. Secondly, to our knowledge, this paper is the first concrete study of parsimony to use Turing machines themselves as the model of computation—rather than (for example) a new programming language proposed by the authors, or a complex on-tape description of Turing machines! We consider it important to use the weakest and most common model of computation for complexity comparisons across different mathematical statements. This is because the more powerful and complex the model of computation used, the more of the complexity of the algorithm can be “shunted” onto the model of computation, and the greater the potential distortion created by the choice of model. As a *reductio ad absurdum*, we could imagine a programming language that included “test the Riemann hypothesis” and “test the consistency of ZFC” as primitive operations. Along a similar vein, very small universal Turing machines typically use a very complex description format for the input machine. In this way, the complexity can again be “shunted” away from the Turing machine itself. By using the “weakest” model of computation that is commonly known, and by requiring that the input tape at initialization be empty, we hope to avoid this pitfall and make it easier to interpret our results in a model-independent way.

³Because Fermat's last theorem and the four color theorem have been proved, their “complexity” is now known to be 1—the minimum number of states in a Turing machine that runs forever.

3 A Turing Machine that Cannot Be Shown to Run Forever Using ZFC

We present a 7,910-state Turing machine whose behavior is *independent of ZFC*; it is not possible to prove that this machine halts or doesn't halt using the axioms of ZFC, assuming that a stronger set theory is consistent. It's therefore impossible to prove the value of $BB(7,910)$ to be any given value without assuming axioms more powerful than ZFC, assuming that ZFC is consistent.

For an explicit listing of this machine, see Appendix C.

We call this machine Z . One way to build this machine would be to start with the axioms of ZFC and apply the inference rules of first-order logic repeatedly in each possible way so as to enumerate every statement ZFC could prove, and to halt if ever a contradiction was found. While this method is conceptually simple, to actually construct such a machine would lead to a huge number of states, because it would require writing a program to manipulate the axioms of ZFC and the inference rules of first-order logic, and then compiling that program all the way down to Turing machine states.

3.1 Friedman's Mathematical Statement

Thankfully, a simpler method exists for creating Z . Friedman [10] was able to derive a graph-theoretic statement whose truth implies the consistency of ZFC, and which will be false if ZFC is inconsistent.⁴ Here is Friedman's statement (the notation will be explained in the rest of this section):

Statement 1. *For all $k, n, r > 0$, every order invariant graph on $[\mathbb{Q}]^{\leq k}$ has a free $\{x_1, \dots, x_r, \text{ush}(x_1), \dots, \text{ush}(x_r)\}$ of complexity $\leq (8knr)!$, each $\{x_1, \dots, x_{(8kni)!}\}$, for $i > 0$ and $(8kni!) \leq r$, reducing $[x_1 \cup \dots \cup x_i \cup \{0, \dots, n\}]^{\leq k}$. [10]*

If s is a set, the operation $(.)^{\leq k}$ refers to the set of all subsets of s with size at most k .

A graph on $[\mathbb{Q}]^{\leq k}$ is an irreflexive symmetric relation on $[\mathbb{Q}]^{\leq k}$. In other words, it can be thought of as a graph whose vertices are elements of $[\mathbb{Q}]^{\leq k}$, and whose edges are undirected, connect pairs of vertices, and never connect vertices to themselves.

A *free* set is a set such that no pair of elements in that set are connected by an edge.

A number of *complexity* at most c refers to a number that can be written as a fraction a/b , where a and b are both integers with absolute value less than or equal to c . A set has complexity at most c if all the numbers it contains have complexity at most c .

An *order invariant graph* is a graph containing a countably infinite number of nodes. In particular, it has one node for each finite set of rational numbers. The only numbers relevant to the statement are numbers of complexity $(8knr)!$ or smaller. In every description of nodes that follows, the term *node* refers both to the object in the order invariant graph and to the set of numbers that it represents.

In an order invariant graph, two nodes (a, b) have an edge between them if and only if each other pair of nodes (c, d) that is *order equivalent* with (a, b) has an edge between them. Two pairs

⁴In fact, Friedman's statement is equivalent to the consistency of SRP ("stationary Ramsey property"), which is a system of axioms more powerful than ZFC. Because SRP is strictly more powerful than ZFC (it in fact consists of ZFC plus some additional axioms), the consistency of SRP implies the consistency of ZFC, and the inconsistency of ZFC implies the inconsistency of SRP.

of nodes (a, b) and (c, d) are *order equivalent* if a and c are the same size and b and d are the same size and if for all $1 \leq i \leq |a|$ and $1 \leq j \leq |b|$, the i -th element of a is less than the j -th element of b if and only if the i -th element of c is less than the j -th element of d .

To give some trivial examples of order invariant graphs: the graph with no edges is order invariant, as is the complete graph. A less trivial example is a graph on $[\mathbb{Q}]^{\leq 2}$, in which each node corresponds to a set of two rational numbers of a given complexity, and there is an edge between two nodes if and only if their corresponding sets a and b satisfy $|a| = |b| = 2$ and $a_1 < b_1 < a_2 < b_2$. (Because edges are undirected in order invariant graphs, such an edge will exist if *either* assignment of the vertices to a and b satisfies the inequality above.)

The *ush()* function takes as input a set and returns a copy of that set with all non-negative numbers in that set incremented by 1.

For vertices x and y , $x \leq_{rlex} y$ if and only if $x = y$ or $x_{|x|-i} < y_{|y|-i}$ where i is the least integer such that $x_{|x|-i} \neq y_{|y|-i}$.⁵ (The \leq_{rlex} operation creates a lexicographic ordering over the vertices, weighting the last and largest elements of those vertices most heavily. Like with lexicographic orderings, if the two vertices are identical but one is longer, the shorter one comes first.)

Finally, a set of vertices X *reduces* a set of vertices Y if and only if for all $y \in Y$, there exists $x \in X$ such that either $x = y$ or $x \leq_{rlex} y$ and an edge exists between x and y .

3.2 Implementation Methods

To create Z , we needed to design a Turing machine that halts if Statement 1 is false, and loops if Statement 1 is true. Such a Turing Machine's behavior is necessarily independent of ZFC, because the truth or falsehood of Statement 1 is independent of ZFC, assuming the consistency of SRP. [10] SRP is an extension of ZFC by certain relatively mild large cardinal hypotheses, and is widely regarded by set theorists as consistent. For more information about SRP, see [13].

To design such a Turing machine, we wrote a Laconic program which encodes Friedman's statement, then compiled the program down to a description of a single-tape, 2-symbol Turing machine. What follows is an extremely brief description of the design of the Laconic program; for the documented Laconic code itself, along with a detailed explanation of the full compilation process, see [25].

Our Laconic program begins by looping over all non-negative values for k , n , and r . For each trio (k, n, r) , our program generates a list N of all numbers of complexity at most $(8knr)!$. These numbers represent the vertices in our putative order invariant graph. Because Laconic does not support floating-point numbers, the list is entirely composed of integers; it is a list of all numbers that can be written in the form $((8knr)!)^{\frac{i}{j}}$, where i and j are integers satisfying $-(8knr)! \leq i \leq (8knr)!$ and $1 \leq j \leq (8knr)!$. (Note that any number that can be expressed in this form is necessarily an integer, because of the large scaling factor in front.)

After we generate N , we generate the nodes in a potential order invariant graph by adding to N all possible lists of k or fewer numbers from N . We call this list of lists V .

We iterate over all binary lists of length $|V|^2$. Any such list E represents a possible set of edges in the graph. To be more precise, we say that an edge exists between node i and node j (represented by V_i and V_j respectively) if and only if $E_{i|V|+j}$ is 1.

For any graph (V, E) , we say that it is "valid" if the following three conditions hold:

⁵Friedman recommended in private communication that we use the \leq_{rlex} comparator to compare vertices, instead of comparing their maximum elements as described in his manuscript.

1. No node has an edge to itself.
2. If an edge exists between node i and node j , an edge also exists between node j and node i .
3. The graph has a free $\{x_1, \dots, x_r, \text{ush}(x_1), \dots, \text{ush}(x_r)\}$, each $\{x_1, \dots, x_{(8kni)!}\}$ reducing $[x_1 \cup \dots \cup x_i \cup \{0, \dots, n\}]^{\leq k}$.

For each list of nodes V , we loop over every possible binary list E , and if no pair (V, E) yields a valid graph, we halt.

When verifying the validity of a graph, checking the first two conditions is trivial, but the third merits further explanation. In order to verify that a given graph (V, E) has a free $\{x_1, \dots, x_r, \text{ush}(x_1), \dots, \text{ush}(x_r)\}$, each $\{x_1, \dots, x_{(8kni)!}\}$ reducing $[x_1 \cup \dots \cup x_i \cup \{0, \dots, n\}]^{\leq k}$, we look at every possible subset of the nodes in V . For each subset, we verify that it has length r , that $\text{ush}(x_1), \dots, \text{ush}(x_r)$ all exist in V , and for each i such that $(8kni)! \leq r$, that $\{x_1, \dots, x_{(8kni)!}\}$ reduces $[x_1 \cup \dots \cup x_i \cup \{0, \dots, n\}]^{\leq k}$. Once we have found such a subset, we know that the third condition is satisfied.

4 A Turing Machine that Encodes Goldbach's Conjecture

We present a 4,888-state Turing machine that *encodes Goldbach's conjecture*; in other words, to know whether this machine halts is to know whether Goldbach's conjecture is true. It is therefore impossible to prove the value of $BB(4,888)$ without simultaneously proving or disproving Goldbach's conjecture.⁶

Recall that Goldbach's conjecture is as follows:

Statement 2. Every even integer greater than 2 can be expressed as the sum of two primes.

Because Goldbach's conjecture is so simple to state, the Laconic program encoding the statement is also quite simple. It can be found in Appendix A. A detailed explanation of the compilation process, documentation for the Laconic language, and an explicit description of this Turing machine are available at [25].

5 A Turing Machine that Encodes Riemann's Hypothesis

We present a 5,372-state Turing machine that *encodes Riemann's hypothesis*; in other words, to know whether this machine halts is to know whether Riemann's hypothesis is true. An explicit description of this machine can be found at [25]

Riemann's hypothesis is traditionally stated as follows:

Statement 3. The Riemann zeta function has its zeros only at the negative even integers and the complex numbers with real part $1/2$.

⁶We note that our tools were primarily meant to encode complex statements into Turing machines, such as the Statement 1. Goldbach's conjecture is quite simple, and as such, it is probably possible to make a dramatically smaller machine through a more direct approach. [1]

5.1 Equivalent Statement

Instead of encoding the Riemann zeta function into a Laconic program, it is simpler to use the following statement, which was shown by Lagarias [5] to be equivalent to the Riemann hypothesis:

Statement 4. *For all integers $n \geq 1$,*

$$\left(\left(\sum_{k \leq \delta(n)} \frac{1}{k} \right) - \frac{n^2}{2} \right)^2 < 36n^3$$

The function $\delta(n)$ used in Statement 4 is defined as follows:

$$\begin{aligned} \eta(j) &= p \text{ if } j = p^k, p \text{ is prime, } k \text{ is a positive integer} \\ \eta(j) &= 1 \text{ otherwise} \\ \delta(x) &= \prod_{n < x} \prod_{j \leq n} \eta(j) \end{aligned}$$

5.2 Implementation Methods

Statement 4 is equivalent to the following statement, which contains only positive integers⁷:

$$l(n) < r(n)$$

for all positive integers n , where

$$\begin{aligned} l(n) &= a(n)^2 + b(n)^2 \\ r(n) &= 36n^3(\delta(n)!)^2 + 2a(n)b(n) \end{aligned}$$

$$\begin{aligned} a(n) &= \sum_{k \leq \delta(n)} \frac{\delta(n)!}{k} \\ b(n) &= \frac{n^2 \delta(n)!}{2} \end{aligned}$$

To check the Riemann hypothesis, our program computes $a(n)$, $b(n)$, $l(n)$, and $r(n)$, in that order, for each possible value of n . If $l(n) \geq r(n)$, our program halts.

6 Laconic

Laconic is a programming language designed to be both user-friendly and easy to compile down to parsimonious Turing machine descriptions.

Laconic is a strongly-typed language that supports recursive functions. Laconic compiles to an intermediate language called TMD. TMD programs are spread across multiple files and grouped into directories. TMD directories are meant to represent sequences of commands that could be

⁷Although it is not immediately obvious, $\frac{\delta(n)!}{k}$ is necessarily an integer for all $k \leq \delta(n)$, and $\frac{\delta(n)!}{2}$ is an integer for all $n > 1$.

given to a multi-tape, 3-symbol Turing machine, using the Turing machine abstraction that allows the machine to read and write from one head at a time.

For an example of a Laconic program, see Appendix A. For an illustration of the compilation process, see Figure 1.

7 TMD

TMD is a programming language designed to help the user describe the behavior of a multi-tape, 3-symbol Turing machine with a function stack. Each tape is infinite in one direction and supports three symbols: `_`, `1`, and `E`. The blank symbol is `_`: that is, `_` is the only symbol that can appear on the tape an infinite number of times. The tape must always have the form $_?(1|E)^+_^\infty$; in other words, each tape must always contain a string of `1`'s and `E`'s of size at least 1, possibly preceded by a `_` symbol, and necessarily followed by an infinite number of copies of the `_` symbol.

What is the purpose of having a language like TMD as an intermediary between Laconic and a description of a single-tape machine? The concept of tapes in a multi-tape Turing machine and the concept of variables in standard imperative programming languages map to one another very nicely. The idea of the Laconic-to-TMD compiler is to encode the value of each variable on one tape. Then, each Laconic command that manipulates the value of one or more variables compiles down to a TMD function call that manipulates the tapes that correspond to those variables appropriately.

As an example, consider the following Laconic command:

```
a=b*c;
```

This Laconic command assigns the value of `b*c` to the variable `a`. It compiles down to the following TMD function call:

```
function BUILTIN_multiply a b c
```

This function call will result in `BUILTIN_multiply` being run on the three tapes `a`, `b`, and `c`. This will cause the symbols on tape `a` to take on a representation of an integer whose value is equal to `bc`.

In turn, the TMD code compiles directly to a string of bits that are written onto the tape at the start of the Turing machine's execution.

A TMD directory consists of three types of files:

1. The **functions** file. This file contains a list of the names of all the functions used by the TMD program. The top function in the file is pushed onto the stack at initialization. When this top function returns, the Turing machine halts.
2. The **initvar** file. This file contains the non-blank symbols that start in each register (or tape) at initialization.
3. Any files used to describe TMD functions. These files all end in a `.tfn` extension and only have any relevance to the compiled program if they show up in the functions file.

8 Compilation and Processing

There are two ways to think about the layout of the tape symbols: with a 4-symbol alphabet ($\{_, 1, H, E\}$, blank symbol $_$), and with a 2-symbol alphabet ($\{a, b\}$, blank symbol a). The 2-symbol alphabet version is the one that's ultimately used for the results in this paper, since we advertised a Turing machine that used only two symbols. However, in nearly all parts of the Turing machine, the 2-symbol version of the machine is a direct translation of the 4-symbol version, according to the following mapping:

$_ \leftrightarrow aa$
 $1 \leftrightarrow ab$
 $H \leftrightarrow ba$
 $E \leftrightarrow bb$

The sections that follow sometimes refer to the **ERROR** state. Transitions to the **ERROR** state should never be taken under any circumstances, and are useful for debugging purposes.

8.1 Concept

A directory of TMD functions is converted at compilation time to a string of bits to be written onto the tape, along with other states designed to interpret these bits. The resulting Turing machine has three main components, or *submachines*:

1. The *initializer* sets up the basic structure of the variable registers and the function stack.
2. The *printer* writes down the binary string that corresponds to the compiled TMD code.
3. The *processor* interprets the compiled binary, modifying the variable registers and the function stack as necessary.

The Turing machine's control flow proceeds from the initializer to the printer to the interpreter. In other words, initializer states point only to initializer states or to printer states, printer states point only to printer states or to interpreter states, and interpreter states point only to interpreter states or the **HALT** state.

This division of labor, while seemingly straightforward, actually constitutes an important idea. The problem of the compiler is to convert a higher-level representation—a machine with many tapes, a larger alphabet, and a function stack—to the lower-level representation of a machine with a single tape, a 2-symbol alphabet and no function stack. The immediately obvious solution, and the one taught in every computability theory class as a proof of the equivalence of different kinds of Turing machines, is to have every “state” in the higher-level machine compile down to many states in the lower-level machine.

While simple, this approach is suboptimal in terms of the number of states. As is nearly always true when designing systems to be parsimonious, the clue that improvement is possible lies in the presence of repetition. Each state transition in the higher-level machine is converted to a group of lower-level states with the same basic structure. Why not instead explain how to perform this conversion exactly once, and then apply the conversion many times?

This idea is at the core of the division of labor described previously. We begin by writing a description of the higher-level machine onto the tape, and then “run” the higher-level machine by reading what is on the tape with a set of states that understands how to interpret the encoded higher-level machine. We refer to this idea as *on-tape processing*.

In this paper, we use TMD as the representation of the higher-level machine.⁸ The printer writes the TMD program onto the tape, and the processor executes it. As a result of using this scheme, we incur a constant *additive* overhead—we have to include the processor in our final Turing machine—but we avoid the constant *multiplicative* overhead required for the naïve scheme.

Is this additive overhead small enough to be worth it? We found that it is. Our implementation of the processor requires 3,860 states. (See Section 8.5 for a detailed breakdown of the state cost by submachine.) In contrast to this additive overhead of 3,860, the naïve approach incurs a large multiplicative overhead that depends in part on how many states must be used to represent each higher-level state transition, and in part on how efficient an encoding scheme can be devised for the on-tape approach. The following table compares the performance of on-tape processing to the performance of an implementation that used the naïve approach. The comparison is shown for three kinds of machines: a machine that halts if and only if Goldbach’s conjecture is false, a machine that halts if and only if the Riemann hypothesis is false, and a machine whose behavior is independent of ZFC.

Program	States (Naïve)	States (On-Tape Processing)
Goldbach	7,902	4,888
Riemann	36,146	5,372
ZFC	340,943	7,910

As can be seen from this table, on-tape interpretation results in huge gains, particularly in large and complex programs.

The subsections that follow describe each of the three submachines—the initializer, the printer, and the processor—in greater detail.

8.2 The Initializer

The initializer starts by writing a counter onto the tape which encodes how many registers there will be in the program. Using the value in that counter, it creates each register, with demarcation patterns between registers, and unique identifiers for each register. Each register’s value begins with the pattern of non-blank symbols laid out in the `initvar` file. The initializer also creates the program counter, which starts at 0, and the function stack, which starts out with only a single function call to the top function in the `functions` file.

Figure 2 is a detailed diagram describing the tape’s state when the initializer passes control to the printer.

⁸Note that instead of TMD, the on-tape processing scheme could be used for any language, assuming the designer provides both a processor and an encoding for that language. We chose TMD because it made the interpreter easy to write, but other minimalist languages, like Unlambda [17], Brainf*ck [20], or Iota and Jot [2], might be good candidates for parsimonious designs, with the additional advantage of being already known to some programmers! Thanks to Luke Schaeffer for this point.

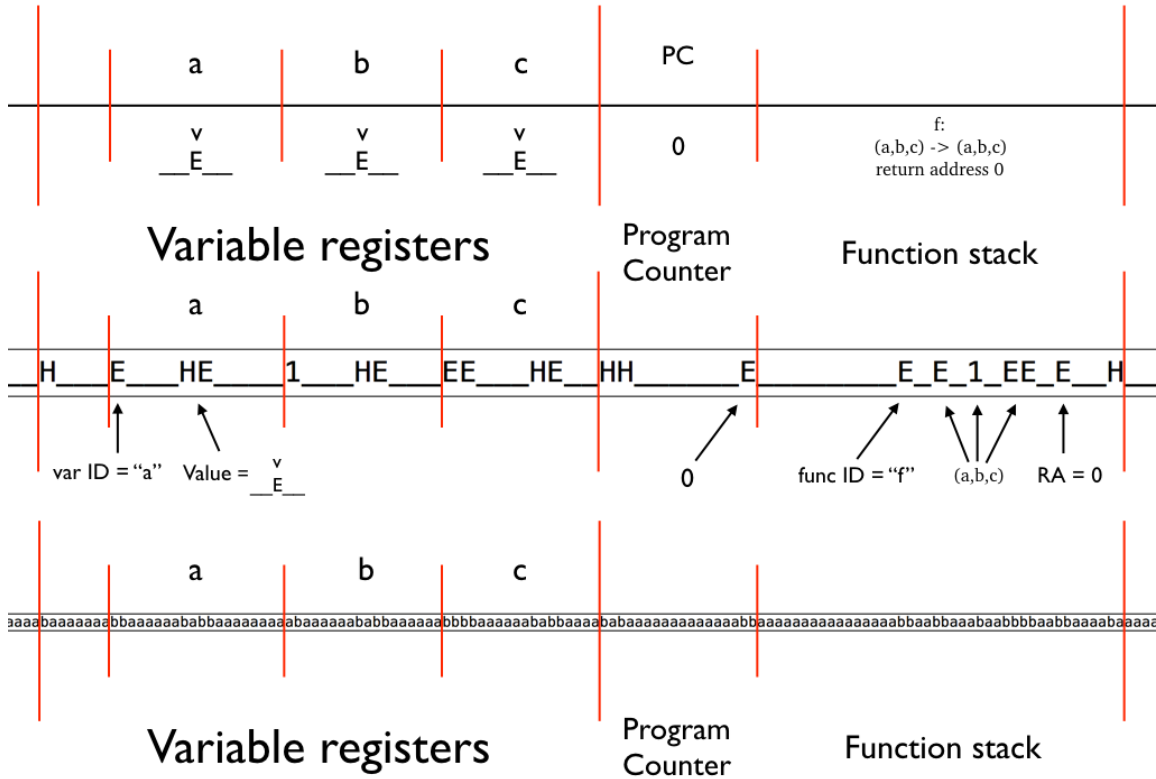


Figure 2: The state of the Turing machine tape after the initializer completes. The TMD program being expressed in Turing machine form is described in full in Appendix B. The top bar is a high-level description of what each part of the Turing machine tape represents. The middle bar is an encoding of the tape in the standard 4-symbol alphabet; the bottom bar is simply the translation of that tape into the 2-symbol alphabet. For a more detailed explanation of how to interpret the tape patterns, see [25].

8.3 The Printer

8.3.1 Specification

The printer writes down a long binary string which encodes the entirety of the TMD program onto the tape.

Figure 3 shows the tape’s state when the printer passes control to the processor.

8.3.2 Introspection

Writing down a long binary string onto a Turing machine tape in a parsimonious fashion is not as straightforward as it might initially appear. The first idea that comes to mind is simply to use one state per symbol, with each state pointing to the next, as shown in Figure 4.

On closer examination, however, this approach is quite wasteful for all but the smallest binary files. Every **a** transition points to the next state in the sequence, and none of the **b** transitions are used at all! Indeed, the only information-bearing part of the state is the single bit contained in the choice of which symbol to write. But in theory, far more information than that could be encoded in each state. In a machine with n states, each state could contain $2(\log_2(n) + 1)$ bits of information, because each of its two transitions could point to any of the n states, and write either an **a** or a **b** onto the tape. Of course, this is only in theory; in practice, to extract the information contained in therefore Turing machine’s states and translate it into bits on the tape is nontrivial.

We will use a scheme originally conceived by Ben-Amram and Petersen [3] and refined further and suggested to us by Luke Schaeffer. It does not achieve the optimal theoretical encoding described above, but is relatively simple to implement and understand, and is within a factor of 2 of optimal for large binary strings. Schaeffer named Turing machines that use this idea *introspective*.

Introspection works as follows. If the binary string contains k bits, then let w be the *word size*. The word size w takes the largest value it can such that $w2^w \leq k$. We can split the binary string into $n_w = \left\lceil \frac{k}{w} \right\rceil$ words of w bits each (we can pad the last word with the blank symbol). In our scheme, each word in the bit-string is represented by a *data state*. Each data state points to the state representing the next word in the sequence for its **a** transition, but which state the **b** transition points to encodes the next word. Every **b** transition points to one of the last 2^w data states, thereby encoding w bits of information.

Of course, the encoding is useless until we specify how to extract the encoded bit-string from the data states. The extraction scheme works as follows. To query the i^{th} data state for the bits it encodes, we run the data states on the string $\mathbf{a}^{i-1}\mathbf{b}\mathbf{a}^\infty$ (a string of $i - 1$ **a**’s followed by a **b** in the i^{th} position). After running the data states on that string, what remains on the tape is the string $\mathbf{b}^{i-1}\mathbf{a}\mathbf{b}^r\mathbf{a}^\infty$, assuming that the i^{th} data state pointed to the r^{th} -to-last data state. Thus, what we’re left with is essentially a unary encoding of the “value” of the word in binary. Thus, the job of the extractor is to set up a binary counter which removes one **b** at a time and increments the counter appropriately. Then, afterward, the extractor reverts the tape back to the form $\mathbf{a}^i\mathbf{b}\mathbf{a}^\infty$, shifts all symbols on the tape over by w bits, and repeats the process. Finally, when the state beyond the last data state sees a **b** on the tape, we know that the process has completed, and we can pass control to the processor. Figure 5 shows the whole procedure.

How much have we gained by using introspection for encoding the program binary, instead of the naïve approach? It depends on how large the program binary is. Using introspection

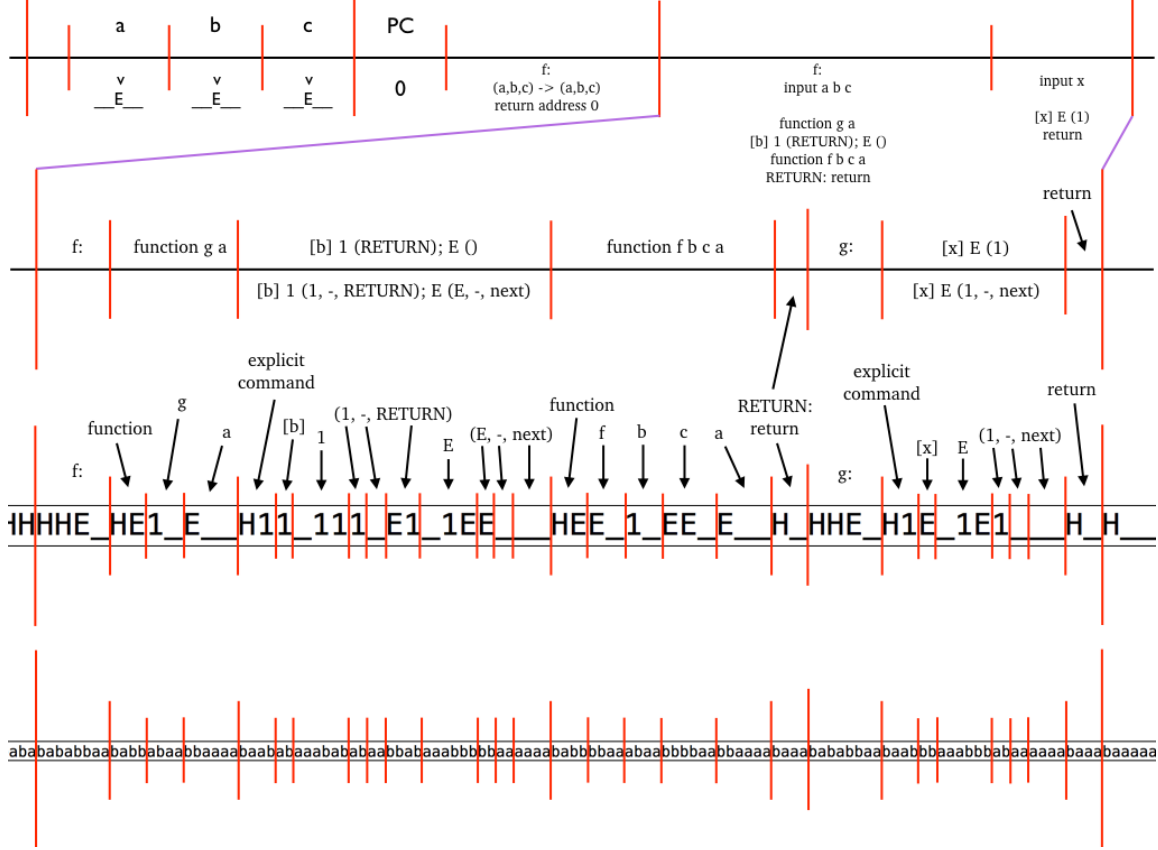


Figure 3: The state of the Turing machine tape after the printer completes. The TMD program being expressed in Turing machine form is described in full in Appendix B. The top bar is a high-level description of the entire tape; unfortunately, at this point there are so many symbols on the tape that it is impossible to see everything at once. For a detailed view of the first two-thirds of the tape (registers, program counter, and stack), see Figure 2. The bottom three bars show a zoomed-in view of the program binary. From the top, the second bar gives a high-level description of what each part of the program binary means; the third bar gives the direct correspondence between 4-symbol alphabet symbols on the tape and their meaning in TMD; the fourth and final bar gives the translation of the third bar into the 2-symbol alphabet. For a more detailed explanation of the encoding of TMD into tape symbols, see [25].

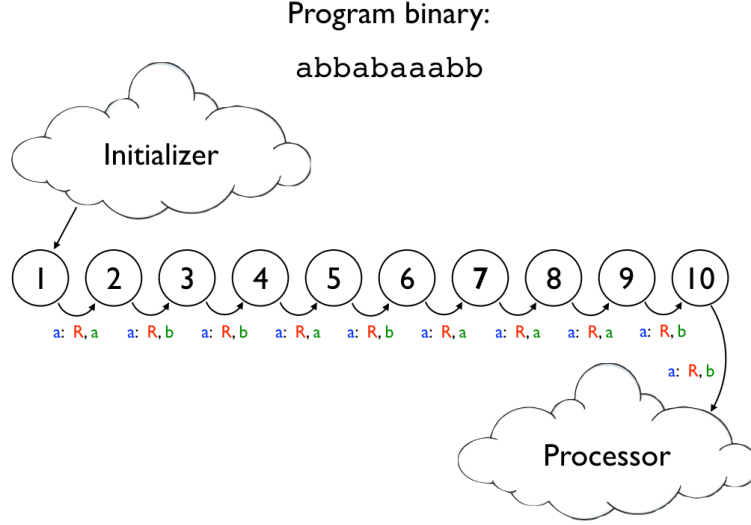


Figure 4: A naïve implementation of the printer. In this example, the hypothetical program is ten bits long, and the printer uses ten states, one for each bit. In the diagram, the blue symbol is the symbol that is read on a transition, the red letter indicates the direction the head moves, and the green symbol indicates the symbol that it written. Note the lack of transitions on reading a **b**; this is because in this implementation, the printer will only ever read the blank symbol, which is **a**, since the head is always proceeding to untouched parts of the tape. It therefore makes no difference what behavior the Turing machine adopts upon reading a **b** in states 1-10 (and therefore **b** transitions are presumed to lead to the **ERROR** state)

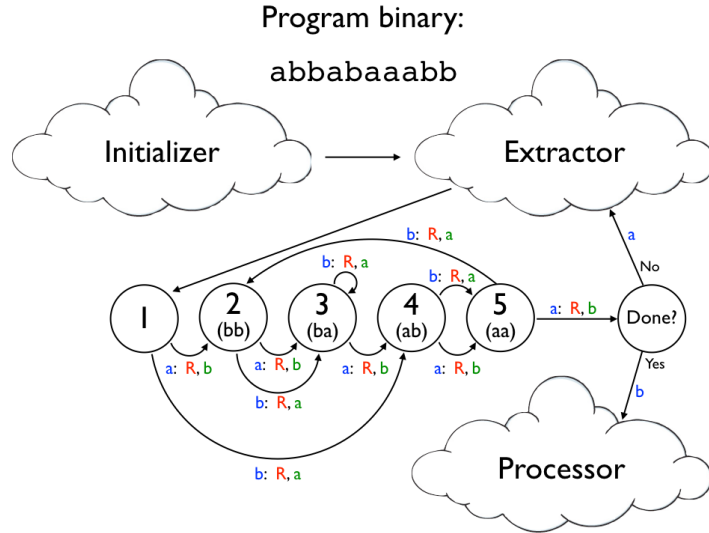


Figure 5: An introspective implementation of the printer. In this example, the hypothetical program is $k = 10$ bits long, and so the word size must be 2 (since $w = 2$ is the largest w such that $w2^w \leq 10$). There are therefore $n_w = \left\lceil \frac{k}{w} \right\rceil = 5$ data states, each encoding two bits. The **b** transitions carry the information about the encoding; note that each one only points to one of the last four data states. The last four data states have in parentheses what word we mean to encode if we point to them.

incurs an $O(\log k)$ *additive* overhead, because we have to include the extractor in our machine. (Our implementation of the extractor takes $10w + 17$ states. It’s possible to build a constant-size extractor, but it’s not worth it for our value of w) In return, we save a *multiplicative* factor of w (which scales with $\log k$) on the number of data states needed.

This is plainly not worth it for the 10-bit example binary shown in Figs. 4 and 5. For that binary, we require 69 additional states for the extractor in order to save 5 data states. For real programs, however, it is worth it, as can be seen from the following table.

Program	Binary Size	w	n_w	Extractor Size	States (Naïve)	States (Introspective)
Example TMD	116	4	29	57	116	86
Goldbach	4,964	9	552	107	4,964	659
Riemann	9,532	10	1,024	117	9,532	1,141
ZFC	38,864	11	3,534	127	38,864	3,661

One minor detail concerns the numbers presented for the Riemann program. Ordinarily, with a binary of size 9,532, we would opt to split the program into 1,060 words of 9 bits each plus a 107-state extractor, since 9 is the greatest w such that $w2^w < 9,532$. But because 9,532 is so close to the “magic number” 10,240, it’s actually more parsimonious to pad the program with copies of the blank symbol until it’s 10,240 bits long, and split it into 1,024 words of 10 bits each plus a 117-state extractor.

8.4 The Processor

The processor’s job is to interpret the code written onto the tape and modify the variable registers and function stack accordingly. The processor does this by the following sequence of steps:

START:

1. Find the function call at the top of the stack. Mark the function f in the code whose ID matches that of the top function call.
2. Read the current program counter. Mark the line of code l in f whose line number matches the program counter.
3. Read l . Depending on what type of command l is, carry out one of the following three lists of tasks.

IF l IS AN EXPLICIT TAPE COMMAND:

1. Read the variable name off l . Index the variable name into the list of variables in the top function on the stack. This list of variables corresponds to the mapping between the function’s local variables and the register names.
2. Match the indexed variable to its corresponding register r . Mark r . Read the symbol s_r to the right of the head marker in that register.
3. Travel back to l , remembering the value of s_r using states. Find and mark the reaction x corresponding to the symbol. See what symbol s_w should be written in response to reading s_r .

4. Travel back to r , remembering the value of s_w using states. Replace s_r with s_w .
5. Travel back to x . See which direction d the head should move in response to reading s_r .
6. Travel back to r , remembering the value of d using states. Move the head marker accordingly.
7. Travel back to x . See if a jump is specified. If a jump is specified, copy the jump address onto the program counter. Otherwise, increment the program counter by 1.
8. Go back to START.

IF l IS A FUNCTION CALL:

1. Write the function's name to the top of the stack.
2. For each variable in the function call, index the variable name into the list of variables in the top function on the stack. This list of variables corresponds to the mapping between the function's local variables and the register names. Push the corresponding register names in the order that they correspond to the variables in the function call.
3. Copy the current program counter to the return address of the newborn function call at the top of the stack.
4. Replace the current program counter with 0 (meaning "read the first line of code").
5. Go back to START.

IF l IS A RETURN STATEMENT:

1. Replace the current program counter with f 's return address.
2. Increment the program counter by 1.
3. Erase the call to f from the top of the stack.
4. Check if the stack is now empty. If so, halt.
5. Go back to START.

8.5 Cost Analysis

It's worthwhile to analyze the relative contributions of the initializer, the printer, and the processor to the machine's final state count. The following table lists the number of states in each submachine for each of the four different TMD programs under discussion.

Program	Initializer	Printer	Processor	Total
Example TMD	349	86	3,860	4,295
Goldbach	369	659	3,860	4,888
Riemann	371	1,141	3,860	5,372
ZFC	389	3,661	3,860	7,910

As can be seen from this table, the processor makes the largest contribution to all four programs. Improving the processor, therefore, is probably the best approach for improving upon the bounds we present. Equally clear, however, is that for programs more complicated than the ones presented here, the cost of the printer will grow almost linearly but the cost of the processor will stay the same. The cost of the initializer grows very slightly with the complexity of programs because of the need to initialize additional registers.

Improving the printer, and with it the TMD and Laconic languages, is probably the best approach for reducing state count for very large and complex programs.

9 Future Work

This paper still leaves a three orders-of-magnitude gap between the smallest n , namely 7,910, for which $BB(n)$ is known to be independent of ZF set theory, and the largest n , namely 4, for which $BB(n)$ is known to be determinable. We regard it as a fascinating problem to pin down the truth here: for example, is it conceivable that $BB(10)$ or even $BB(6)$ might be independent of ZF? If so, that would arguably force a qualitative change in our understanding of the Gödel incompleteness phenomenon—showing that incompleteness from strong set theories rears its head for much simpler arithmetical questions than had previously been known.

A more immediate question is how much further Z 's state count can be reduced. It seems very likely that Z 's state count can, will (assuming further inquiry) be reduced. There are a variety of ways that this might happen, but the most promising approach is probably to adapt the processor-printer model to use a better language than TMD—a language whose processor contains fewer states, whose binary encodings of programs will require fewer bits, or ideally, both. A few ideas have been suggested [1], but the most popular suggestion is some sort of Lambda-Calculus-based processor.

Other future work might involve further use of our Laconic language to upper-bound the ‘complexities’ of mathematical statements and algorithms, in as standardized and model-independent a way as possible. Perhaps Laconic could be used to measure the complexity of other well-known conjectures, or even to compare different algorithms for solving the same problem to each other (e.g., to try to quantify the notion that an insertion sort is simpler than a merge sort)!

10 Acknowledgements

We thank Prof. Harvey Friedman for having done the crucial theoretical work that made this project feasible. Prof. Friedman was endlessly available over email, and provided us with detailed clarifications when we needed them.

We thank Luke Schaeffer for his early help, as well as his help designing introspective Turing machines.

We thank Alex Arkhipov for introducing us to the term “code golfing.”

Supported by an Alan T. Waterman Award from the National Science Foundation, under grant no. 1249349.

References

- [1] Aaronson, S. “The 8000th Busy Beaver number eludes ZF set theory: new paper by Adam Yedidia and me.” May 3, 2016. <http://www.scottaaronson.com/blog/?p=2725#comments> [Scott Aaronson publicized a preprint of our results on his blog, and many of his readers offered helpful comments and suggestions for future improvements.]
- [2] Barker, C. “Iota and Jot: the Simplest Languages?” <http://semarch.linguistics.fas.nyu.edu/barler/Iota/> [A website describing the Iota and Jot programming languages]
- [3] Ben-Amram, A., Petersen, H. “Improved Bounds for Functions Related to Busy Beavers” *Theory of Computing Systems* 35, 1-11 (2002)
- [4] Brady, A.H. “Solution of the Non-computable ‘Busy Beaver’ game for $k = 4$.” Abstracts for: ACM Computer Science Conference (Washington, DC, February 18-20, 1975), p. 27, ACM, 1975.
- [5] Browder, F. “Mathematical Developments Arising from Hilbert Problems.” American Mathematical Society. Volume 28, Part 1.
- [6] Calude, C., Calude, E. “Evaluating the Complexity of Mathematical Problems: Part 1,” “Evaluating the Complexity of Mathematical Problems: Part 2.” *Complex Systems* 18, pp. 387-401. 2010.
- [7] Chaitin, G. “The Limits of Mathematics.” pp. 79. 1994.
- [8] Cloudy176, Wythagoras. “A good bound for $S(7)$?” 2014. http://googology.wikia.com/wiki/User_blog:Wythagoras/A_good_bound_for_S%287%29%3F
- [9] Deedlit11, Wythagoras. “Okay, more Turing machines.” 2013. http://googology.wikia.com/wiki/User_blog:Deedlit11/Okay,_more_Turing_machines
- [10] Friedman, H. “Order Invariant Graphs and Finite Incompleteness.” <https://u.osu.edu/friedman.8/files/2014/01/FIiniteSeqInc062214a-v9w7q4.pdf>
- [11] Personal communications with H. Friedman.
- [12] Friedman, H. “Order Theoretic Equations, Maximality, and Incompleteness.” June 7, 2014. <http://u.osu.edu/friedman.8/foundational-adventures/downloadable-manuscripts#78>.
- [13] Friedman H. “The Upper Shift Kernel Theorems.” October 9, 2010. <https://u.osu.edu/friedman.8/files/2014/01/KernStruThm100910-1lu0b8v.pdf>
- [14] Gödel, K. “The Consistency of the Axiom of Choice and of the Generalized Continuum-Hypothesis with the Axioms of Set Theory.” Published in 1940 by the Princeton University Press. *Annals of Mathematics Studies*.

- [15] Koza, J. “Spontaneous Emergence of Self-Replicating and Evolutionarily Self-Improving Computer Programs.” in *Artificial Life III* (SFI Studies in the Sciences of Complexity, vol. XVII), C. G. Langton, Ed. Reading, MA: Addison-Wesley. pp. 225-262. 1994.
- [16] Lin, S., Rado, T. “Computer Studies of Turing Machine Problems.” Published in *Journal of the ACM*, Volume 12, Issue 2, April 1965. Pages 196-212.
- [17] Madore, D. “The Unlambda Programming Language.” <http://www.madore.org/~david/programs/unlambda/>
[A website describing the Unlambda programming language]
- [18] Marxen, H., Buntrock, J. “Attacking the Busy Beaver 5.” *Bull EATCS*, Vol. 40, pp. 247-251. 1990.
- [19] Marxen, H. <http://www.drb.insel.de/~heiner/BB/>
[A list of the known busy beaver values]
- [20] Müller, U. “Brainfuck.” <http://www.muppetlabs.com/~breadbox/bf/>
[A website describing the Brainf*ck programming language]
- [21] Pargellis, A. “The Spontaneous Generation of Digital ‘Life.’” *Physica D*, 91, 86-96. 1996.
- [22] Rado, T. “On Non-Computable Functions.” *Bell System Technical Journal*, 41: 3. May 1962 pp 877-884.
- [23] Schoenfield, J. “The Problem of Predicativity.” *Essays on the foundations of mathematics*, Y. Bar-Hillel et al., eds., pp. 132-142. 1961.
- [24] Smith, A. “Universality of Wolfram’s 2, 3 Turing Machine.” Submitted for the Wolfram 2, 3 Turing Machine Research Prize. <http://www.wolframscience.com/prizes/tm23/TM23Proof.pdf>
- [25] Yedidia, A. <https://github.com/adamyedidia/parsimony>
[A link to a GitHub repository containing all programs and Turing machines related to this paper, with accompanying documentation.]
- [26] <http://codegolf.stackexchange.com/>
[A place where programmers go for recreational code golfing]

Appendices

A Example Laconic Program: Goldbach’s Conjecture

The following is an example Laconic program, which compiles down to the Turing machine G mentioned in Section 4 (which halts if and only Goldbach’s Conjecture is false).

```
func zero(x) {
  x = 0;
  return;
}
```

```

func one(x) {
    x = 1;
    return;
}

func incr(x) {
    x = x + 1;
    return;
}

/* Computes x modulo y */
func modulus(x, y, out) {
    out = x;

    while (out >= y) {
        out = out - y;
    }

    return;
}

func assignXtoYminusX(x, y) {
    x = y - x;
    return;
}

/* Figures out if x is prime, and puts the output in y */
/* Does not modify x, modifies y */
func isPrime(x, h, y) {
    if (x == 1) {
        zero(y);
        return;
    }

    y = 2;

    while (x > y) {
        modulus(x, y, h);

        if (h == 0) {
            zero(y);
            return;
        }
        incr(y);
    }

    return;
}

int evenNumber;
int primeCounter;
int isThisOnePrime;
int foundSum;
int h;

evenNumber = 2;
one(foundSum);

while (foundSum) {
    zero(foundSum);
    evenNumber = evenNumber + 2;
    one(primeCounter);

    while (primeCounter < evenNumber) {

```



```

    isPrime(primeCounter, h, isThisOnePrime);

    if (isThisOnePrime) {
        assignXtoYminusX(primeCounter, evenNumber);
        isPrime(primeCounter, h, isThisOnePrime);
        assignXtoYminusX(primeCounter, evenNumber);

        if (isThisOnePrime) {
            print evenNumber;
            print primeCounter;

            one(foundSum);
        }
    }

    incr(primeCounter);
}

halt;

```

For detailed documentation of the Laconic programming language, see [25]. To find this file specifically, navigate to `parsimony/src/laconic/laconic_files/goldbach.lac` at [25].

B Example TMD Program

The following is an example TMD directory, which compiles down to a binary string to be written on a Turing machine tape. It's the example used in illustrations throughout this paper, most notably in the example compilation shown in Figs. 2 and 3. The program calls itself recursively three times until the starting symbol on each tape, E, is replaced with a 1, at which point the program halts.

This TMD directory is called `example_tmd_dir`, and contains four files: `f.tmd`, `g.tmd`, `initvar`, and `functions`.

```

f.tmd:
input a b c

// Recursively writes a 1 on every tape.

function g a
[b] 1 (RETURN); E ()
function f b c a
RETURN: return

g.tmd:
input x

// Writes a 1 on the input tape.

[x] E (1)
return

functions:

f
g

initvar:

```

For detailed documentation of the TMD programming language, see [25]. To find this directory specifically, navigate to `parsimony/src/tmd/tmd_dirs/example_tmd_dir/` at [25].

C Explicit Listing of Z

A description of a single state in Z

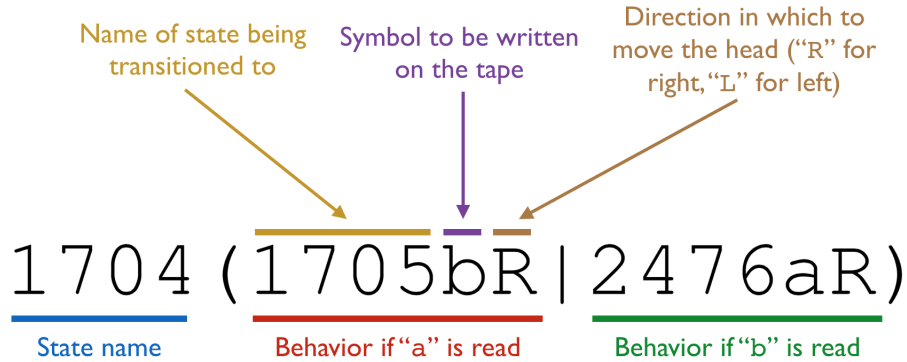


Figure 6: This figure explains how to read a description of a single state. Note that “ERROR–” or “HALT–” denote transitions to the **ERROR** or **HALT** states, respectively (no further information is provided because what symbol is written and which direction the head moves are at that point irrelevant).

We present below an explicit listing of Z . For a more easily readable version of Z , complete with descriptive state names, see [25].

We ran this Turing machine for 10,000,000,000 steps (more than half a day on our simulators) and within that time it did not halt. We note, however, that Z was designed for parsimony rather than efficiency, and that this “experiment” is of little consequence! We similarly ran a Turing machine designed to test the conjecture that all perfect squares are less than 5, and it ran for 2,895,083,899 steps (a couple hours on our simulator) before it found the counterexample 9 and halted.

Figure 6 explains how to interpret the description shown below. In addition, note the following:

1. The tape has a 2-symbol alphabet, with tape symbols $\{a, b\}$ and blank symbol \mathbf{a} (in other words, \mathbf{a} is the only symbol that can appear an infinite times on the tape).
2. The start state of Z is 0000.

3. Z will never transition to the ERROR state. Any transition to the ERROR state could be replaced by a transition to any other state (including HALT) and the Turing machine's behavior would remain identical.

4. Z contains only one transition to the HALT state, out of state 7862.

0000(0001b ERRR--)	0001(0004b ERRR--)	0002(0003b ERRR--)	0003(0012a 0012b)	0004(0005a ERRR--)	0005(0006b ERRR--)	0006(0007b ERRR--)	0007(0008b ERRR--)	0008(0009b ERRR--)	0009(0010b ERRR--)
0010(0011b ERRR--)	0011(0002b ERRR--)	0012(0013a ERRR--)	0013(0014a 0014b)	0014(0005a ERRR--)	0015(0007b 0057b)	0016(0017b 0057b)	0017(0018b ERRR--)	0018(0019a ERRR--)	0019(0020a 0020b)
0020(0021b ERRR--)	0022(0022a 0022b)	0022(0023a ERRR--)	0023(0024a 0024b)	0024(0025a ERRR--)	0025(0067a 0067b)	0026(0027a 0032b)	0027(0028a 0030b)	0028(0029a 0029b)	0029(0026a 0026b)
0030(0031a 0031b)	0031(0022b 0022b)	0032(0033a 0033b)	0033(0034a 0034b)	0034(0035a 0035b)	0035(0036a 0036b)	0036(0026a 0026b)	0037(0038a 0038b)	0038(0044a 0039a)	0039(0040b 0040b)
0040(0049a 0049b)	0041(0003a 0043a)	0042(0043a 0043a)	0043(0037a 0037b)	0044(0004a 0048b)	0045(0006a 0046a)	0046(0047a 0047a)	0047(0048a 0049b)	0048(0059a 0059b)	0049(0050a 0051b)
0050(0049a 0049b)	0051(0052a 0049b)	0052(0053a 0054b)	0053(0052a 0052b)	0054(0055a 0052b)	0055(0056a 0056a)	0056(0012a 0012b)	0057(0058a ERRR--)	0058(0059a ERRR--)	0059(0060a ERRR--)
0060(0061a 0061b)	0061(0062a ERRR--)	0062(0063a 0063b)	0063(0064a ERRR--)	0064(0065a 0065b)	0065(0066a ERRR--)	0066(0016a 0016b)	0067(0068a ERRR--)	0068(0069a ERRR--)	0069(0070a ERRR--)
0070(0026a 0026b)	0071(0027a 0075b)	0072(0027a 0075b)	0073(0047a 0074b)	0074(0005a 0078b)	0075(0058a 0076b)	0076(0077a 0077b)	0077(0078a 0078b)	0078(0079a 0082b)	0079(0080a 0078b)
0080(0081b 0081b)	0081(0083a 0083b)	0082(0077a 0078b)	0083(0084a 0085b)	0084(0085a 0083b)	0085(0086a 0083b)	0086(0087a 0087b)	0087(0088a 0088b)	0088(0089a 0094b)	0089(0090a 0092b)
0090(0091a 0091b)	0091(0091a 0091b)	0092(0091a 0091b)	0093(0091a 0091b)	0094(0091a 0091b)	0095(0091a 0091b)	0096(0091a 0091b)	0097(0091a 0091b)	0098(0091a 0091b)	0099(0100a 0100b)
0100(0101a 0101b)	0101(0102a 0102b)	0102(0102a 0102b)	0103(0104a 0104b)	0104(0101a 0101b)	0105(0106a 0108b)	0106(0107a 0107a)	0107(0108a 0108b)	0108(0109a 0109b)	0109(0110a 0110b)
0110(0111a 0111b)	0111(0112b 0114b)	0112(0113b 0113b)	0113(0130a 0130b)	0114(0115b 0115b)	0115(0116a 0110b)	0116(0117a 0117b)	0117(0118a 0118a)	0118(0119a 0119b)	0119(0120a 0120b)
0120(0121a 0121b)	0121(0122b 0122b)	0122(0131a 0131b)	0123(0142a 0142b)	0124(0143a 0143b)	0125(0144a 0144b)	0126(0127a 0127b)	0127(0128a 0128b)	0128(0129a 0129b)	0129(0130a 0130b)
0130(0131a 0131b)	0131(0132a 0134b)	0132(0133a 0133b)	0133(0130a 0130b)	0134(0131a 0135b)	0135(0088a 0088b)	0136(0088a 0137b)	0137(0138a 0138b)	0138(0088a 0088b)	0139(0140a 0143b)
0140(0139a 0141b)	0141(0142a 0142b)	0142(0146a 0146b)	0143(0147a 0147b)	0144(0148a 0145b)	0145(0146a 0146b)	0146(0147a 0150b)	0147(0148a 0146b)	0148(0149a 0149b)	0149(0177a 0077b)
0150(0151a 0151b)	0151(0152a 0152b)	0152(0153a 0153b)	0153(0154a 0154b)	0154(0155a 0155b)	0155(0156a 0156b)	0156(0157a 0157b)	0157(0158a 0158b)	0158(0159a 0159b)	0159(0160a 0160b)
0160(0157a 0157b)	0161(0162a 0162b)	0162(0166a 0166b)	0163(0197a 0164b)	0164(0165a 0165b)	0165(0166a 0166b)	0166(0167a 0172b)	0167(0168a 0171b)	0168(0169b 0169b)	0169(0171a 0171b)
0170(0171a 0171b)	0171(0166a 0166b)	0172(0173a 0175b)	0173(0174a 0174b)	0174(0166a 0166b)	0175(0176a 0176b)	0176(0166a 0166b)	0177(0178a 0183b)	0178(0179a 0181b)	0179(0180a 0180b)
0180(0171a 0171b)	0181(0168a 0168b)	0182(0168a 0168b)	0183(0168a 0184b)	0184(0168a 0185b)	0185(0168a 0186b)	0186(0187a 0192a)	0187(0188a 0190b)	0188(0189a 0190b)	0189(0187a 0187b)
0190(0191a 0191b)	0191(0193a 0195b)	0192(0193a 0195b)	0193(0194a 0194b)	0194(0195a 0195b)	0195(0196a 0196b)	0196(0197a 0197b)	0197(0198a 0198b)	0198(0197a 0197b)	0199(0200a 0197b)
0200(0201a 0204b)	0201(0202a 0197b)	0202(0203a 0203b)	0203(0205a 0205b)	0204(0206a 0206b)	0205(0206a ERRR--)	0206(0207a 0207b)	0207(0208a ERRR--)	0208(0209a 0209b)	0209(0210a ERRR--)
0210(0219a 0219b)	0211(0212a ERRR--)	0212(0213a ERRR--)	0213(0214a ERRR--)	0214(0215a ERRR--)	0215(0216a ERRR--)	0216(0217a ERRR--)	0217(0218a ERRR--)	0218(0219a ERRR--)	0219(0220a ERRR--)
0220(0221a 0221b)	0221(0222a ERRR--)	0222(0223a 0223b)	0223(0224a ERRR--)	0224(0225a ERRR--)	0225(0226a ERRR--)	0226(0227a ERRR--)	0227(0228a ERRR--)	0228(0229a ERRR--)	0229(0230a ERRR--)
0230(0231a 0231b)	0231(0232a ERRR--)	0232(0211a 0211b)	0233(0234a ERRR--)	0234(0235a ERRR--)	0235(0236a ERRR--)	0236(0237a ERRR--)	0237(0238a ERRR--)	0238(0239a ERRR--)	0239(0240a ERRR--)
0240(0241a ERRR--)	0241(0242a ERRR--)	0242(0243a ERRR--)	0243(0244a ERRR--)	0244(0245a ERRR--)	0245(0246a ERRR--)	0246(0247a ERRR--)	0247(0248a ERRR--)	0248(0249a ERRR--)	0249(0250a ERRR--)
0250(0251a 0251b)	0251(0252a ERRR--)	0252(0253a 0253b)	0253(0254a ERRR--)	0254(0255a 0255b)	0255(0256a ERRR--)	0256(0257a 0257b)	0257(0258a ERRR--)	0258(0259a 0259b)	0259(0260a ERRR--)
0260(0261a 0261b)	0261(0262a ERRR--)	0262(0215a 0215b)	0263(0264a 0269b)	0264(0265a 0265b)	0265(0266a 0266b)	0266(0267a 0267b)	0267(0268a 0268b)	0268(0269a 0269b)	0269(0270a 0270b)
0270(0271a 0271b)	0271(0272a 0274b)	0272(0273a 0273b)	0273(0274a 0274b)	0274(0275a 0275b)	0275(0276a 0276b)	0276(0277a 0277b)	0277(0278a 0278b)	0278(0279a 0279b)	0279(0280a 0280b)
0280(0274a 0274b)	0281(0282a 0285b)	0282(0283a ERRR--)	0283(0284a 0284b)	0284(0285a 0285b)	0285(0286a ERRR--)	0286(0287a ERRR--)	0287(0300a 0300b)	0288(0301a 0301b)	0289(0288a 0288b)
0290(0291a 0291b)	0291(0292a 0293b)	0292(0293a 0293b)	0293(0294a 0294b)	0294(0295a 0295b)	0295(0296a 0296b)	0296(0297a ERRR--)	0297(0298a ERRR--)	0298(0299a ERRR--)	0299(0263a 0263b)
0300(0301a 0301b)	0301(0302a 0302b)	0302(0303a 0303b)	0303(0304a 0304b)	0304(0305a 0305b)	0305(0306a 0306b)	0306(0307a 0307b)	0307(0308a 0308b)	0308(0309a 0309b)	0309(0310a 0310b)
0310(0311a 0311b)	0311(0312a 0312b)	0312(0313a 0313b)	0313(0314a 0314b)	0314(0315a 0315b)	0315(0316a 0316b)	0316(0317a 0317b)	0317(0318a 0318b)	0318(0319a 0319b)	0319(0320a 0320b)
0320(0321a 0321b)	0321(0322a 0322b)	0322(0323a 0323b)	0323(0324a 0324b)	0324(0325a 0325b)	0325(0326a 0326b)	0326(0327a 0327b)	0327(0328a 0328b)	0328(0329a 0329b)	0329(0330a 0330b)
0330(0331a 0331b)	0331(0332a 0332b)	0332(0333a 0333b)	0333(0334a 0334b)	0334(0335a 0335b)	0335(0336a 0336b)	0336(0337a 0337b)	0337(0338a 0338b)	0338(0339a 0339b)	0339(0340a 0340b)
0340(0341a 0341b)	0341(0342a 0342b)	0342(0343a 0343b)	0343(0344a 0344b)	0344(0345a 0345b)	0345(0346a 0346b)	0346(0347a 0347b)	0347(0348a 0348b)	0348(0349a 0349b)	0349(0350a 0350b)
0350(0351a 0351b)	0351(0352a 0352b)	0352(0353a 0353b)	0353(0354a 0354b)	0354(0355a 0355b)	0355(0356a 0356b)	0356(0357a 0357b)	0357(0358a 0358b)	0358(0359a 0359b)	0359(0360a 0360b)
0360(0361a 0361b)	0361(0362a 0362b)	0362(0363a 0363b)	0363(0364a 0364b)	0364(0365a 0365b)	0365(0366a 0366b)	0366(0367a 0367b)	0367(0368a 0368b)	0368(0369a 0369b)	0369(0370a 0370b)
0370(0360a 0360b)	0371(0372a 0375b)	0372(0373a 0373b)	0373(0374a 0374b)	0374(0375a 0375b)	0375(0376a 0376b)	0376(0377a 0377b)	0377(0378a 0378b)	0378(0379a 0379b)	0379(0380a ERRR--)
0380(0381a ERRR--)	0381(0382a 0382b)	0382(0383a ERRR--)	0383(0384a ERRR--)	0384(0385a ERRR--)	0385(0386a ERRR--)	0386(0387a ERRR--)	0387(0388a ERRR--)	0388(0389a ERRR--)	0389(0390a ERRR--)
0390(0391a ERRR--)	0391(0392a ERRR--)	0392(0393a ERRR--)	0393(0394a ERRR--)	0394(0395a ERRR--)	0395(0396a ERRR--)	0396(0397a ERRR--)	0397(0398a ERRR--)	0398(0399a ERRR--)	0399(0400a ERRR--)
0400(0401a ERRR--)	0401(0402a ERRR--)	0402(0403a ERRR--)	0403(0404a ERRR--)	0404(0405a ERRR--)	0405(0406a ERRR--)	0406(0407a ERRR--)	0407(0408a ERRR--)	0408(0409a ERRR--)	0409(0410a ERRR--)
0410(0411a ERRR--)	0411(0412a ERRR--)	0412(0413a ERRR--)	0413(0414a ERRR--)	0414(0415a ERRR--)	0415(0416a ERRR--)	0416(0417a ERRR--)	0417(0418a ERRR--)	0418(0419a ERRR--)	0419(0420a ERRR--)
0420(0421a ERRR--)	0421(0422a ERRR--)	0422(0423a ERRR--)	0423(0424a ERRR--)	0424(0425a ERRR--)	0425(0426a ERRR--)	0426(0427a ERRR--)	0427(0428a ERRR--)	0428(0429a ERRR--)	0429(0430a ERRR--)
0430(0431a ERRR--)	0431(0432a ERRR--)	0432(0433a ERRR--)	0433(0434a ERRR--)	0434(0435a ERRR--)	0435(0436a ERRR--)	0436(0437a ERRR--)	0437(0438a ERRR--)	0438(0439a ERRR--)	0439(0440a ERRR--)
0440(0441a ERRR--)	0441(0442a ERRR--)	0442(0443a ERRR--)	0443(0444a ERRR--)	0444(0445a ERRR--)	0445(0446a ERRR--)	0446(0447a ERRR--)	0447(0448a ERRR--)	0448(0449a ERRR--)	0449(0450a ERRR--)
0450(0451a ERRR--)	0451(0452a ERRR--)	0452(0453a ERRR--)	0453(0454a ERRR--)	0454(0455a ERRR--)	0455(0456a ERRR--)	0456(0457a ERRR--)	0457(0458a ERRR--)	0458(0459a ERRR--)	0459(0460a ERRR--)
0460(0461a ERRR--)	0461(0462a ERRR--)	0462(0463a ERRR--)	0463(0464a ERRR--)	0464(0465a ERRR--)	0465(0466a ERRR--)	0466(0467a ERRR--)	0467(0468a ERRR--)	0468(0469a ERRR--)	0469(0470a ERRR--)
0470(0471a ERRR--)	0471(0472a ERRR--)	0472(0473a ERRR--)	0473(0474a ERRR--)	0474(0475a ERRR--)	0475(0476a ERRR--)	0476(0477a ERRR--)	0477(0478a ERRR--)	0478(0479a ERRR--)	0479(0480a ERRR--)
0480(0481a ERRR--)	0481(0482a ERRR--)	0482(0483a ERRR--)	0483(0484a ERRR--)	0484(0485a ERRR--)	0485(0486a ERRR--)	0486(0487a ERRR--)	0487(0488a ERRR--)	0488(0489a ERRR--)	0489(0490a ERRR--)
0490(0491a ERRR--)	0491(0492a ERRR--)	0492(0493a ERRR--)	0493(0494a ERRR--)	0494(0495a ERRR--)	0495(0496a ERRR--)	0496(0497a ERRR--)	0497(0498a ERRR--)	0498(0499a ERRR--)	0499(0500a ERRR--)
0500(0501a ERRR--)	0501(0502a ERRR--)	0502(0503a ERRR--)	0503(0504a ERRR--)	0504(0505a ERRR--)	0505(0506a ERRR--)	0506(0507a ERRR--)	0507(0508a ERRR--)	0508(0509a ERRR--)	0509(0510a ERRR--)
0510(0511a ERRR--)	0511(0512a ERRR--)	0512(0513a ERRR--)	0513(0514a ERRR--)	0514(0515a ERRR--)	0515(0516a ERRR--)	0516(0517a ERRR--)	0517(0518a ERRR--)	0518(0519a ERRR--)	0519(0520a ERRR--)
0520(0521a ERRR--)	0521(0522a ERRR--)	0522(0523a ERRR--)	0523(0524a ERRR--)	0524(0525a ERRR--)	0525(0526a ERRR--)	0526(0527a ERRR--)	0527(0528a ERRR--)	0528(0529a ERRR--)	0529(0530a ERRR--)
0530(0531a ERRR--)	0531(0532a ERRR--)	0532(0533a ERRR--)	0533(0534a ERRR--)	0534(0535a ERRR--)	0535(0536a ERRR--)	0536(0537a ERRR--)	0537(0538a ERRR--)	0538(0539a ERRR--)	0539(0540a ERRR--)
0540(0541a ERRR--)	0541(0542a ERRR--)	0542(0543a ERRR--)	0543(0544a ERRR--)	0544(0545a ERRR--)	0545(0546a ERRR--)	0546(0547a ERRR--)	0547(0548a ERRR--)	0548(0549a ERRR--)	0549(0550a ERRR--)
0550(0551a ERRR--)	0551(0552a ERRR--)	0552(0553a ERRR--)	0553(0554a ERRR--)	0554(0555a ERRR--)	0555(0556a ERRR--)	0556(0557a ERRR--)	0557(0558a ERRR--)	0558(0559a ERRR--)	0559(0560a ERRR--)
0560(0561a ERRR--)	0561(0562a ERRR--)	0562(0563a ERRR--)	0563(0564a ERRR--)	0564(0565a ERRR--)	0565(0566a ERRR--)	0566(0567a ERRR--)	0567(0568a ERRR--)	0568(0569a ERRR--)	0569(0570a ERRR--)
0570(0571a ERRR--)	0571(0572a ERRR--)	0572(0573a ERRR--)	0573(0574a ERRR--)	0574(0575a ERRR--)	0575(0576a ERRR--)	0576(0577a ERRR--)	0577(0578a ERRR--)	0578(0579a ERRR--)	0579(0580a ERRR--)
0580(0581a ERRR--)	0581(0582a ERRR--)	0582(0583a ERRR--)							

“

20

30

31

7480(7475aL|7475bL) 7481(7482aL|7484bL) 7482(7483aL|7483bL) 7483(7323aL|7323bL) 7484(7485aL|7485bL) 7485(7475aL|7475bL) 7486(7487aR|7492bR) 7487(7488aL|7490bL) 7488(7489aL|7489bL) 7489(7486aL|7486bL)
7490(7491aL|7491bL) 7491(7486aL|7486bL) 7492(7493aL|7495bL) 7493(7494aL|7494bL) 7494(7334aL|7334bL) 7495(7496aL|7496bL) 7496(7486aL|7486bL) 7497(7498aR|7503bR) 7498(7499aL|7501bL) 7499(7500aL|7500bL)
7500(7497aL|7497bL) 7501(7502aL|7502bL) 7502(7497aL|7497bL) 7503(7506aR|7504bL) 7504(7505aL|7505bL) 7505(7497aL|7497bL) 7506(7507aR|7510bR) 7507(7506aR|7509bL) 7508(7509aR|7509bR) 7509(7513aL|7513bL)
7510(ERROR-|7511bL) 7511(7512aR|7512bR) 7512(7513aL|7513bL) 7513(7514aR|7515bR) 7514(7516aR|7513bR) 7515(7513aR|7513bR) 7516(7517bR|ERROR-) 7517(7518aR|7518bR) 7518(7519aR|7520bR) 7519(7518aR|7518bR)
7520(7521aR|7518bR) 7521(7522aR|7523bR) 7522(7521aR|7521bR) 7523(7524aL|7521bR) 7524(7525aR|7525aR) 7525(7526aR|7526bR) 7526(7527aR|7532bR) 7527(7528aL|7530bL) 7528(7529aR|7529bR) 7529(7535aL|7535bL)
7530(7531aR|7531bR) 7531(7046aL|7046bL) 7532(ERROR-|7533bL) 7533(7534aR|7534bR) 7534(7046aL|7046bL) 7535(7536aR|7541bR) 7536(7537aL|7539bL) 7537(7538aL|7539bL) 7538(7535aL|7535bL) 7539(7540aL|7540bL)
7540(7535aL|7535bL) 7541(7542aL|7544bL) 7542(7543aL|7543bL) 7543(7546aL|7546bL) 7544(7545aL|7545bL) 7545(7535aL|7535bL) 7546(7547aR|7552bR) 7547(7548aL|7550bL) 7548(7549aL|7549bL) 7549(7546aL|7546bL)
7550(7551aL|7551bL) 7551(7546aL|7546bL) 7552(7553aL|7555bL) 7553(7554aR|7554aR) 7554(7557aL|7557bL) 7555(7556aL|7556bL) 7556(7546aL|7546bL) 7557(7558aR|7563bR) 7558(7559aL|7561bL) 7559(7560aL|7560bL)
7560(7557aL|7557bL) 7561(7562aR|7562bR) 7562(7566aL|7566bL) 7563(ERROR-|7564bL) 7564(7565aR|7565bR) 7565(7566aL|7566bL) 7566(7567aR|7572bR) 7567(7568aL|7570bL) 7568(7569bR|7569bR) 7569(7577aL|7577bL)
7570(7571aL|7571bL) 7571(7566aL|7566bL) 7572(7573aL|7575bL) 7573(7574aL|7574bL) 7574(7566aL|7566bL) 7575(7576aL|7576bL) 7576(7566aL|7566bL) 7577(7578aR|7583bR) 7578(7579aL|7581aL) 7579(7580aL|7580bL)
7580(7618aL|7618bL) 7581(7582bL|7582bL) 7582(7586aL|7586bL) 7583(7594aR|7584aL) 7584(7585bL|7585bL) 7585(7588aL|7588bL) 7586(ERROR-|7587aR) 7587(7590bR|ERROR-) 7588(ERROR-|7589bR) 7589(7592bR|ERROR-)
7590(ERROR-|7591bR) 7591(7599aR|7599bR) 7592(ERROR-|7593bR) 7593(7604aR|7604bR) 7594(7595aR|7595bR) 7595(7594aR|7594bR) 7596(7597aL|7594aR) 7597(7598aR|7598aR) 7598(7643aR|7643bR) 7599(7600aR|7601bR)
7600(7599aR|7599bR) 7601(7602bL|7599bR) 7602(7603aR|7603aR) 7603(7643aR|7643bR) 7604(7605aR|7605bR) 7605(7604aR|7604bR) 7606(7607bL|7604bR) 7607(7608bR|7608bR) 7608(7643aR|7643bR) 7609(7610aR|7615bR)
7610(7611aL|7615bL) 7611(7612aL|7612bL) 7612(7609aL|7609bL) 7613(7614aL|7614bL) 7614(7609aL|7609bL) 7615(7677aR|7616bL) 7616(7617aL|7617bL) 7617(7609aL|7609bL) 7618(ERROR-|7619bR) 7619(7620bL|ERROR-)
7620(7621aL|7621bL) 7621(7622aL|7622bL) 7622(7623aR|7629bR) 7623(7624aL|7626aL) 7624(7625aR|7625bR) 7625(7631aL|7631bL) 7626(7627aL|7627bL) 7627(7622aL|7622bL) 7628(ERROR-|7629aL) 7629(7630aL|7630bL)
7630(7622aL|7622bL) 7631(7632aR|7633bR) 7632(7631aR|7631bR) 7633(7631aR|7631bR) 7634(7635aR|7638bR) 7635(7636bL|7634bR) 7636(7637aR|7637aR) 7637(7661aR|7661bR) 7638(7639bL|7641bL) 7639(7640aR|7640aR)
7640(7643aR|7643bR) 7641(7642aR|7642aR) 7642(7652aR|7652bR) 7643(7644aR|7649bR) 7644(7645aL|7647aL) 7645(7646bR|7646bR) 7646(7661aR|7661bR) 7647(7648bR|7648bR) 7648(7634aR|7634bL) 7649(7670aR|7650aL)
7650(7651bR|7651bR) 7651(7652aR|7652bR) 7652(7653aR|7658bR) 7653(7654bL|7656bL) 7654(7655bR|7655bR) 7655(7661aR|7661bR) 7656(7657aR|7657bR) 7657(7634aR|7634bR) 7658(7659bL|7659bR) 7659(7660bR|7660bR)
7660(7643aR|7643bR) 7661(7662aR|7665bR) 7662(7661aR|7663aL) 7663(7664aR|7664aR) 7664(7634aR|7634bR) 7665(7666aL|7666bL) 7666(7667aR|7667aR) 7667(7643aR|7643bR) 7668(7669aR|7669aR) 7669(7652aR|7652bR)
7670(7671aR|7676bR) 7671(7672aL|7674aL) 7672(7673bR|7673bR) 7673(7661aR|7661bR) 7674(7675bR|7675bR) 7675(7634aR|7634bR) 7676(7679aR|7677aL) 7677(7678bR|7678bR) 7678(7652aR|7652bR) 7679(7680bR|ERROR-)
7680(7681aL|7681bL) 7681(7682aR|7685bR) 7682(7683aL|ERROR-) 7683(7684aR|7684aR) 7684(7685bL|7685bL) 7685(7686aL|ERROR-) 7686(7687aL|7687bL) 7687(7681aL|7681bL) 7688(7689aR|7694bR) 7689(7690aL|7692bL)
7690(7691aL|7691bL) 7691(7688aL|7688bL) 7692(7693aL|7693bL) 7693(7688aL|7688bL) 7694(7695aR|7697bL) 7695(7696aL|7696bL) 7696(7699aL|7699bL) 7697(7698aL|7698bL) 7698(7688aL|7688bL) 7699(7700aR|7705bR)
7700(7701aL|7703bL) 7701(7702aL|7702bL) 7702(7699aL|7699bL) 7703(7704aL|7704bL) 7704(7699aL|7699bL) 7705(7706aL|7708bL) 7706(7707aL|7707aL) 7707(7710aL|7710bL) 7708(7709aL|7709bL) 7709(7699aL|7699bL)
7710(7711aR|7716bR) 7711(7712aL|7714bL) 7712(7713aL|7713bL) 7713(7711aL|7711bL) 7714(7715aR|7715bR) 7715(7719aL|7719bL) 7716(ERROR-|7717bL) 7717(7718aR|7718bR) 7718(7719aL|7719bL) 7719(7720aR|7725bR)
7720(7721aL|7723bL) 7721(7722bL|7722bL) 7722(7730aL|7730bL) 7723(7724aL|7724bL) 7724(7719aL|7719bL) 7725(7726aL|7728bL) 7726(7727aL|7727bL) 7727(7719aL|7719bL) 7728(7729aL|7729bL) 7729(7719aL|7719bL)
7730(7731aR|7736bR) 7731(7732aL|7734bL) 7732(7733aL|7733bL) 7733(7730aL|7730bL) 7734(7735aL|7735bL) 7735(7730aL|7730bL) 7736(7737aL|7739bL) 7737(7738aL|7738aL) 7738(7741aL|7741bL) 7739(7740aL|7740bL)
7740(7730aL|7730bL) 7741(7742bL|ERROR-) 7742(7743aL|7743bL) 7743(7744aR|7749bR) 7744(7745aL|7747aL) 7745(7746aR|7746bR) 7746(7752aL|7752bL) 7747(7748aL|7748aL) 7748(7743aL|7743bL) 7749(ERROR-|7750aL)
7750(7751aL|7751aL) 7751(7743aL|7743bL) 7752(7753aR|7754bR) 7753(7752aR|7752bR) 7754(7755aR|7752bR) 7755(7756aR|7757bR) 7756(7755aR|7755bR) 7757(7758aL|7758bL) 7758(7759aR|7759aR) 7759(7760aR|7760bR)
7760(7761aR|7766bR) 7761(7762aL|7764aL) 7762(7763aR|7763bR) 7763(7743aL|7743bL) 7764(7765bL|7765bL) 7765(7769aL|7769bL) 7766(ERROR-|7767aL) 7767(7768bL|7768bL) 7768(7780aL|7780bL) 7769(7770aR|7775bR)
7770(7771aL|7773bL) 7771(7772aL|7772bL) 7772(7769aL|7769bL) 7773(7774aL|7774bL) 7774(7769aL|7769bL) 7775(7776aL|7778bL) 7776(7777aL|7777aL) 7777(7791aL|7791bL) 7778(7793aL|7793bL) 7779(7793aL|7793bL)
7780(7781aR|7786bR) 7781(7782aL|7783bL) 7782(7783aL|7783bL) 7783(7780aL|7780bL) 7784(7787aL|7787bL) 7785(7788aL|7788bL) 7786(7787aL|7789bL) 7787(7788aL|7789bL) 7788(7781aL|7818bL) 7789(7790aL|7790bL)
7790(7780aL|7780bL) 7791(7792aR|7797bR) 7792(7793bL|7795bL) 7793(7794aR|7794aR) 7794(7827aL|7827bL) 7795(7796aL|7796bL) 7796(7791aL|7791bL) 7797(ERROR-|7798bL) 7798(7799aL|7799aL) 7799(7800aL|7800bL)
7800(7801aR|7806bR) 7801(7802bL|7804bL) 7802(7803bR|7803bR) 7803(7827aL|7827bL) 7804(7805bL|7805bL) 7805(7791aL|7791bL) 7806(ERROR-|7807bL) 7807(7808aL|7808bL) 7808(7810aR|7815bR) 7809(7810aR|7815bR)
7810(7811bL|7813bL) 7811(7812aR|7812aR) 7812(7830aL|7830bL) 7813(7814aL|7814bL) 7814(ERROR-|7820bL) 7815(7816aL|7816bL) 7816(7817aL|7817aL) 7817(7818aL|7818bL) 7818(7819aR|7824bR) 7819(7820bL|7822bL)
7820(7821bR|7821bR) 7821(7830aL|7830bL) 7822(7823bL|7823bL) 7823(7809aL|7809bL) 7824(ERROR-|7825bL) 7825(7826aL|7826bL) 7826(7818aL|7818bL) 7827(7828aR|7829bR) 7828(7827aR|7827bR) 7829(7833aR|7833aR)
7830(7831aR|7832bR) 7831(7830aL|7830bR) 7832(7833aR|7833bR) 7833(7834aR|7835bR) 7834(7833aR|7833bR) 7835(7836bL|7833bR) 7836(7837aR|7837aR) 7837(7760aR|7760bR) 7838(7839aR|7840bR) 7839(7838aR|7838bR)
7840(7841bL|7838bR) 7841(7842bR|7842bR) 7842(7760aR|7760bR) 7843(7844aR|7849bR) 7844(7845aL|7847aL) 7845(7846aL|7846bL) 7846(7852aL|7852bL) 7847(7848aL|7848aL) 7848(7843aL|7843bL) 7849(ERROR-|7850aL)
7850(7851aL|7851aL) 7851(7843aL|7843bL) 7852(7853aR|7858bR) 7853(7854aL|7856aL) 7854(7855aL|7855bL) 7855(7861aL|7861bL) 7856(7857aL|7857aL) 7857(7843aL|7843bL) 7858(ERROR-|7859aL) 7859(7860aL|7860aL)
7860(7843aL|7843bL) 7861(7862aR|7863bR) 7862(BL-|7864aR) 7863(ERROR-|7864aR) 7864(7865aR|7870bR) 7865(7866aL|7866bL) 7866(7867aL|7867aL) 7867(7864aL|7864bL) 7868(7869aL|7869bL) 7869(7864aL|7864bL)
7870(7871aL|7873bL) 7871(7872aL|7872aL) 7872(7875aL|7875bL) 7873(7874aL|7874bL) 7874(7864aL|7864bL) 7875(7876aL|7881bL) 7876(7877bL|7879bL) 7877(7878bR|7878bR) 7878(7884aL|7884bL) 7879(7880bL|7880bL)
7880(7875aL|7875bL) 7881(ERROR-|7882bL) 7882(7883aR|7883aR) 7883(7884aL|7884bL) 7884(7885aR|7886bR) 7885(7887aR|7884bR) 7886(7884aR|7884bR) 7887(7888aR|7889bR) 7888(7887aR|ERROR-) 7889(ERROR-|7890bL)
7890(7891aR|7891bR) 7891(7892aL|7892bL) 7892(7893aR|7894bR) 7893(7895aR|7895bR) 7894(7892aR|7892bR) 7895(7896aR|7897bR) 7896(7898aR|7899bR) 7897(7892aR|7892bR) 7898(7899aR|7902bR) 7899(7900aL|7892bR)
7900(7901aR|7901bR) 7901(7903aL|7903bL) 7902(7892aR|7892bR) 7903(7904aR|7905bR) 7904(7903aR|ERROR-) 7905(ERROR-|7906bL) 7906(7907aL|7907bL) 7907(7908aL|7908bL) 7908(7909bL|ERROR-) 7909(4381aL|4381bL)