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

April 8, 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 7,870 by presenting an explicit description of a 7,870-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. We also present a 4,888-state Turing machine G that halts if and only if there is a counterexample to Goldbach's conjecture, and a 5,372-state Turing machine R that halts if and only if the Riemann hypothesis is false. To create G, R, and Z, we develop and use a higher-level language, Laconic, which is much more convenient than direct state manipulation.

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.

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. 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.

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

Q is the set of k *states* $\{q_0, q_1, \ldots, q_{k-2}, q_{k-1}\}$
$\Gamma = \{a, b\}$ is the set of *tape alphabet symbols*
`a` is the *blank symbol*
$\Sigma =$ is the set of *input symbols*
$\delta = Q \times \Gamma \to (Q \cup F) \times \Gamma \times \{L, R\}$ is the *transition function*
q_0 is the *start state*
$F = \{\text{ACCEPT}, \text{REJECT}, \text{ERROR}\}$ is the set of *halting transitions.*

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) and the symbol read off the tape (an element of Γ).

[1] While 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, [9] and ZFC and ZF prove exactly the same arithmetical statements (which include, among other things, statements about whether Turing machines halt). [18]

It outputs three instructions: what state to enter (an element of Q), 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 that causes the Turing machine to halt. While having three possible halting transitions is not necessary for our purposes, being able to differentiate between three different types of halting (ACCEPT, REJECT, 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. [17]

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 [11] [3]:

$$BB(1) = 1$$

$$BB(2) = 6$$

$$BB(3) = 21$$

$$BB(4) = 107$$

For $BB(5)$ and $BB(6)$, only lower bounds are known: $BB(5) \geq$ 47,176,870, and $BB(6) \geq 7.4 \times 10^{36,534}$. Researchers have worked on pinning down the value of $BB(5)$ exactly, and some consider it to be possibly within reach. A summary of the current state of human knowledge about Busy Beaver values can be found at [14].

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 7,870, 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 7,870 states. Therefore, we will never be able to prove the value of $BB(7{,}870)$ 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 7,870 , 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 [20]. 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. [2] 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 known to be stronger than ZFC (and whose consistency, therefore, would imply the consistency of ZFC). [7]

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 what 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 (for whom Chaitin's constant, Ω, is named) raised the work of this paper as a possible problem for mathematicians to pursue in his book *The Limits of Mathematics*. He wrote, "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 in concerned, just to see how many bits of complexity mathematicians normally assume.... If you programmed ZF, you'd get a really sharp

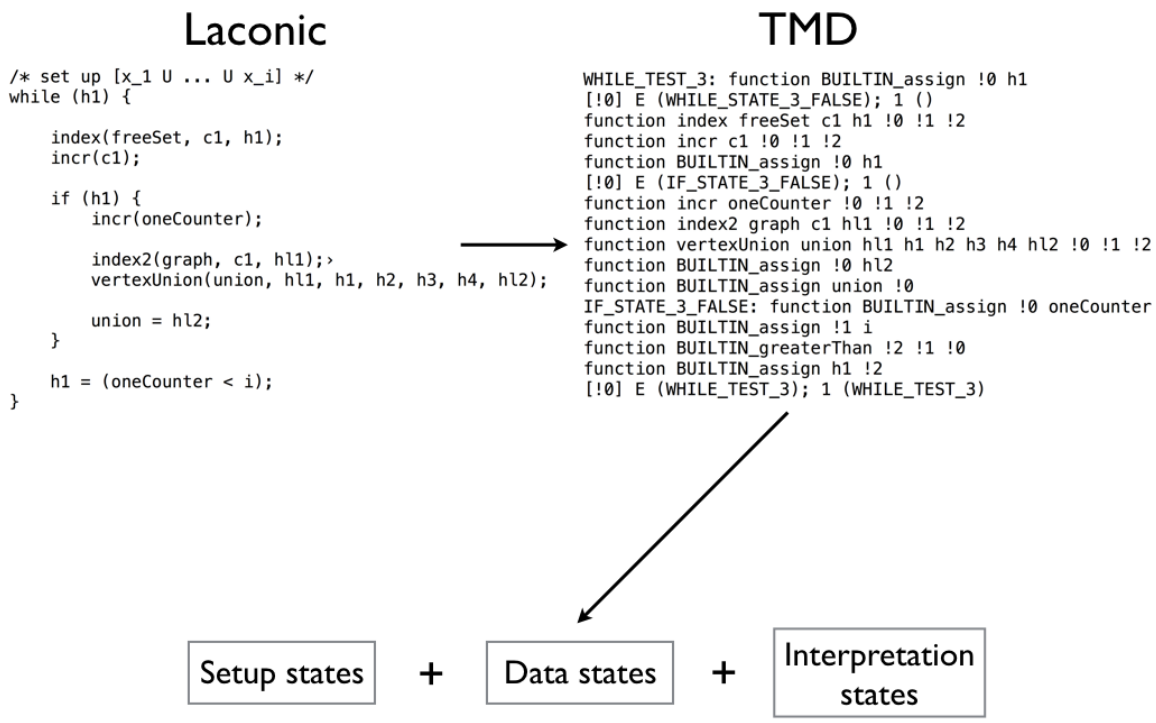


Figure 1: A visual overview of the compilation process.

incompleteness result. It wouldn't say that you can get at most $H(ZF) + 15328$ bits of Ω, it would say, perhaps, at most 96000 bits! We'd have a much more definite incompleteness theorem." We didn't program ZF set theory in LISP, but we programmed it in an even simpler language, and answered Chaitin's call for an explicit number of bits to attach to the complexity of ZF set theory. [6]

This paper is not the first to attempt to quantify the complexity of arithmetical statements. Calude and Calude [5] 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.[2] In addition, Koza [10] and Pargellis [16] 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."

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 as the model of computation—rather than (for example) a new programming language proposed by the authors! 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. By using the "weakest" model of computation that is commonly known, and one which is generally accepted as the mathematical basis of algorithms, we hope to avoid this pitfall and make it easier to interpret our results in a model-independent way.

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

We present a 7,870-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 slightly stronger set theory is consistent. It's therefore impossible to prove the value of $BB(7{,}870)$ 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.

[2] Because 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.

3.1 Friedman's Mathematical Statement

Thankfully, a simpler method exists for creating Z. Friedman [7] was able to derive a graph-theoretic statement whose truth implies the consistency of ZFC, and which will be false if ZFC is inconsistent.[3] 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 $\{x_1, \ldots, x_r, ush(x_1), ..., ush(x_r)\}$ of complexity $\leq (8knr)!$, each $\{x_1, \ldots, x_{(8kni)!}\}$ reducing $[x_1 \cup \cdots \cup x_i \cup \{0, \ldots, n\}]^{\leq k}$. [7]*

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 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 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 $[\mathbb{Q}]^2$, 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 $a_1 < b_1 < a_2 < b_2$. (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 $x_i < y_i$ where i is least such that $x_i \neq y_i$.

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, a slightly more powerful set theory. [7]

[3] In 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.

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 [19].

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 $(((8knr)!)!)((8kni)!)/((8knj)!)$, 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:

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 $\{x_1, \ldots, x_r, \mathrm{ush}(x_1), ..., \mathrm{ush}(x_r)\}$, each $\{x_1, \ldots, x_{(8kni)!}\}$ reducing $[x_1 \cup \cdots \cup x_i \cup \{0, \ldots, n\}]^{\leq k}$.

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 $\{x_1, \ldots, x_r, ush(x_1), ..., ush(x_r)\}$, each $\{x_1, \ldots, x_{(8kni)!}\}$ reducing $[x_1 \cup \cdots \cup x_i \cup \{0, \ldots, n\}]^{\leq k}$, we look at every possible subset of the nodes in V. For each subset, we verify that it has length r, that $\mathrm{ush}(x_1), ..., \mathrm{ush}(x_r)$ all exist in V, and for each i such that $(8kni)! \leq r$, that $\{x_1, \ldots, x_{(8kni)!}\}$ reduces $[x_1 \cup \cdots \cup x_i \cup \{0, \ldots, n\}]^{\leq k}$. Once we have found such a subset, we know that the third conditon 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.

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 [19].

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 [19]

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.

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 [4] 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:

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

5.2 Implementation Methods

Statement 4 is equivalent to the following statement, which contains only positive integers[4]:

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

for all positive integers n, where

$$l(n) = a(n)^2 + b(n)^2$$
$$r(n) = 36n^3(\delta(n)!)^2 + 2a(n)b(n)$$

.

$$a(n) = \sum_{k \leq \delta(n)} \frac{\delta(n)!}{k}$$

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

$$b(n) = \frac{n^2 \delta(n)!}{2}$$

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 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 $\texttt{\_}?(\texttt{1}|\texttt{E})^+\texttt{\_}^\infty$; 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 `a` to the value of `b*c`. 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-`_` symbols that start in each register 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 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.[5] 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	7,870

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.

[5]Note 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 [12], Brainf*ck [15], or Iota and Jot [1], 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.

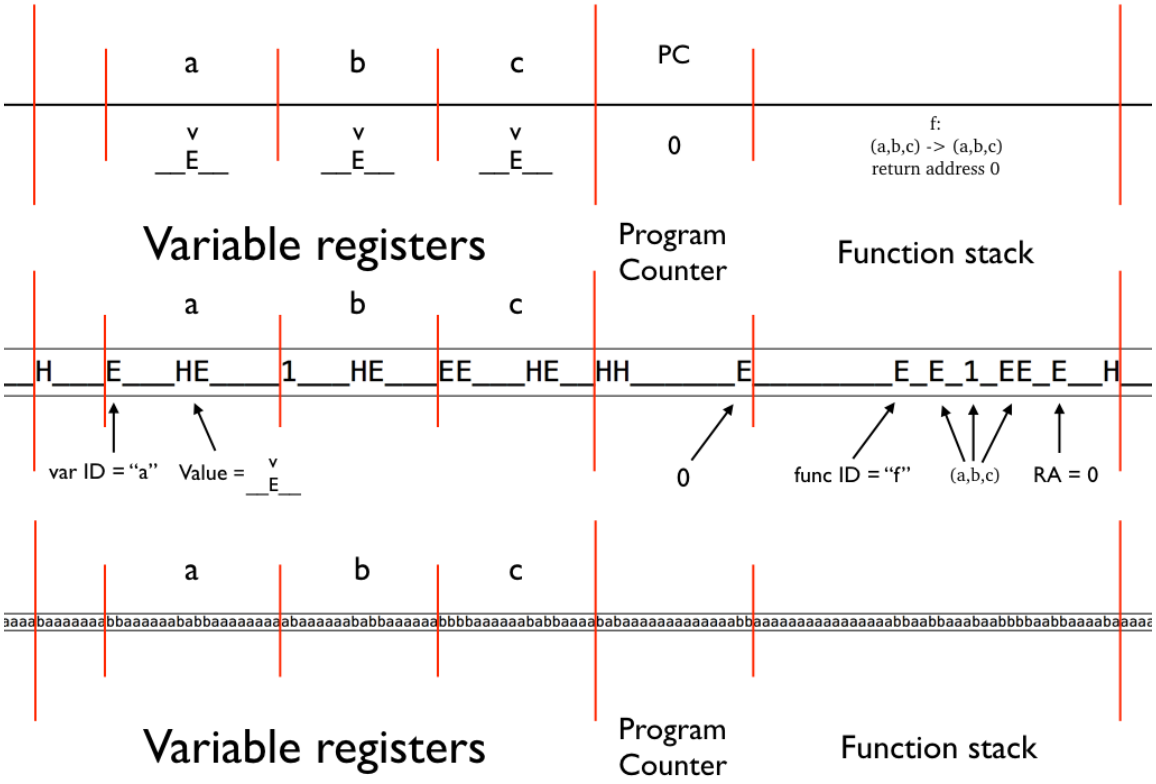


Figure 2: The state of the Turing machine tape after the initializer completes. The TMD program being expressed in Turing machine form is described in full in Appendix B. The top bar is a high-level description of what each part of the Turing machine tape represents. The middle bar is an encoding of 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 [19].

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-`_` 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.

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 the 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 [2] 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*. 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 copies of 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 $\mathtt{a}^{i-1}\mathtt{ba}^{\infty}$ (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 $\mathtt{b}^{i-1}\mathtt{ab}^{r}\mathtt{a}^{\infty}$, 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 $\mathtt{a}^{i}\mathtt{ba}^{\infty}$, 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 incurs

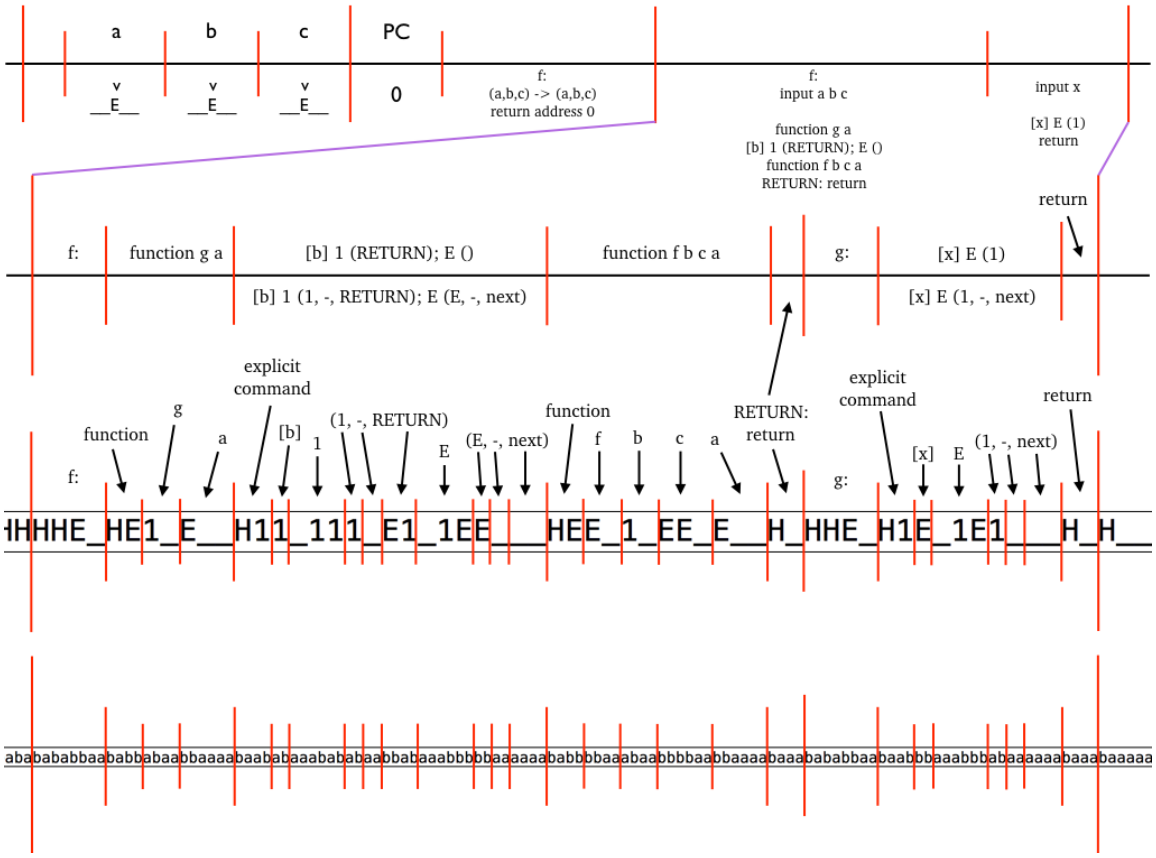


Figure 3: The state of the Turing machine tape after the printer completes. The TMD program being expressed in Turing machine form is described in full in Appendix B. The top bar is a high-level description of the entire tape; unfortunately, at this point there are so many symbols on the tape that it 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 [19].

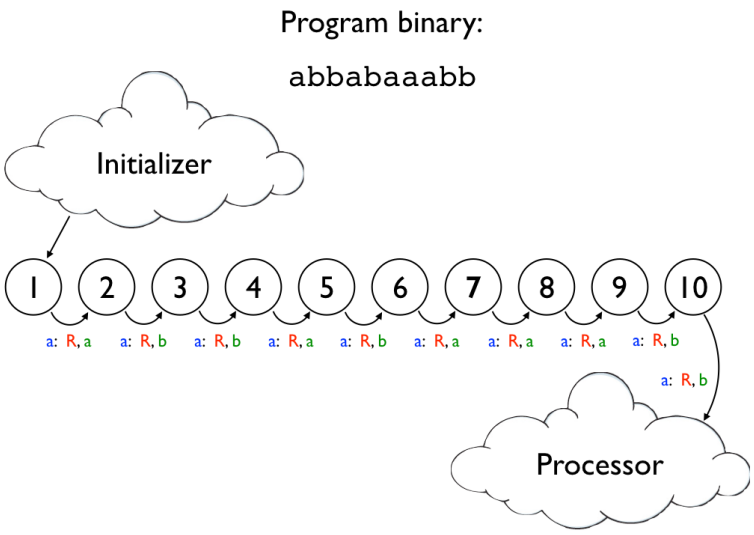


Figure 4: A naïve implementation of the printer. In this example, the hypothetical program is ten bits long, and the printer uses ten states, one for each bit. In the diagram, the blue symbol is the symbol that is read on a transition, the red letter indicates the direction the head moves, and the green 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)

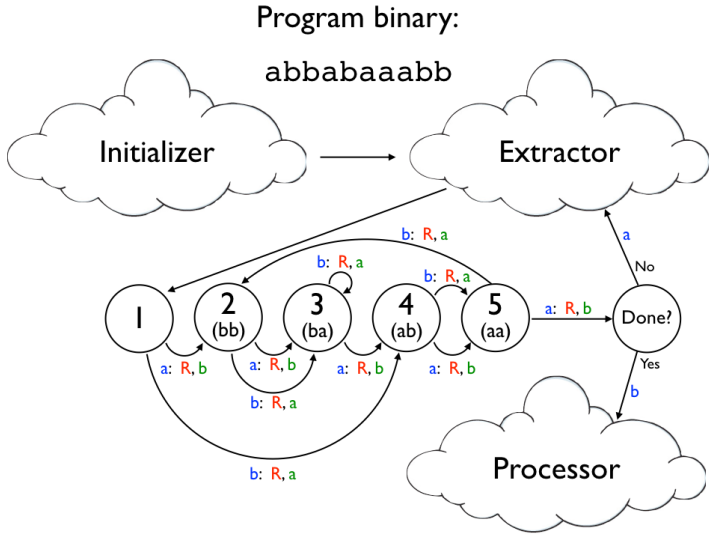


Figure 5: An introspective implementation of the printer. In this example, the hypothetical program is $k = 10$ bits long, and so the word size must be 2 (since $w = 2$ is the largest w such that $w2^w \leq 10$). There are therefore $n_w = \left\lceil \frac{k}{w} \right\rceil = 5$ data states, each encoding 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.

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.) 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	38,426	11	3,494	127	38,426	3,621

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	3,621	3,860	7,870

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

How much further can Z's state count be reduced? Without a significant new insight, further order-of-magnitude reductions seem unlikely via tweaks to our existing system: note that such tweaks would need to reduce the sizes of both the printer and the processor by an order of magnitude. However, improvement is possible by (for example) a redesigned processor, combined with a simpler independent mathematical statement than Friedman's.

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 immediate and 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] 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]

[2] Ben-Amram, A., Petersen, H. "Improved Bounds for Functions Related to Busy Beavers" Theory of Computing Systems 35, 1-11 (2002)

[3] 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.

[4] Browder, F. “Mathematical Developments Arising from Hilbert Problems.” American Mathematical Society. Volume 28, Part 1.

[5] 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.

[6] Chaitin, G. “The Limits of Mathematics.” pp. 79. 1994.

[7] Friedman, H. “Order Invariant Graphs and Finite Incompleteness.” `https://u.osu.edu/friedman.8/files/2014/01/FIiniteSeqInc062214a-v9w7q4.pdf`

[8] Friedman, H. “Order Theoretic Equations, Maximality, and Incompleteness.” June 7, 2014. `http://u.osu.edu/friedman.8/foundational-adventures/downloadable-manuscripts` #78.

[9] 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.

[10] 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.

[11] 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.

[12] Madore, D. “The Unlambda Programming Language.” `http://www.madore.org/~david/programs/unlambda/`
[A website describing the Unlambda programming language]

[13] Marxen, H., Buntrock, J. “Attacking the Busy Beaver 5.” Bull EATCS, Vol. 40, pp. 247-251. 1990.

[14] Marxen, H. `http://www.drb.insel.de/~heiner/BB/`
[A list of the known busy beaver values]

[15] Müller, U. “Brainfuck.” `http://www.muppetlabs.com/~breadbox/bf/`
[A website describing the Brainf*ck programming language]

[16] Pargellis, A. “The Spontaneous Generation of Digital ‘Life.’” Physica D, 91, 86-96. 1996.

[17] Rado, T. “On Non-Computable Functions.” Bell System Technical Journal, 41: 3. May 1962 pp 877-884.

[18] Schoenfield, J. “The Problem of Predicativity.” Essays on the foundations of mathematics, Y. Bar-Hillel et al., eds., pp. 132-142. 1961.

[19] 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.]

[20] 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 [19]. To find this file specifically, navigate to `parsimony/src/laconic/laconic_files/goldbach.lac` at [19].

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`:

```
E
```

For detailed documentation of the TMD programming language, see [19]. To find this directory specifically, navigate to `parsimony/src/tmd/tmd_dirs/example_tmd_dir/` at [19].

C Explicit Listing of Z

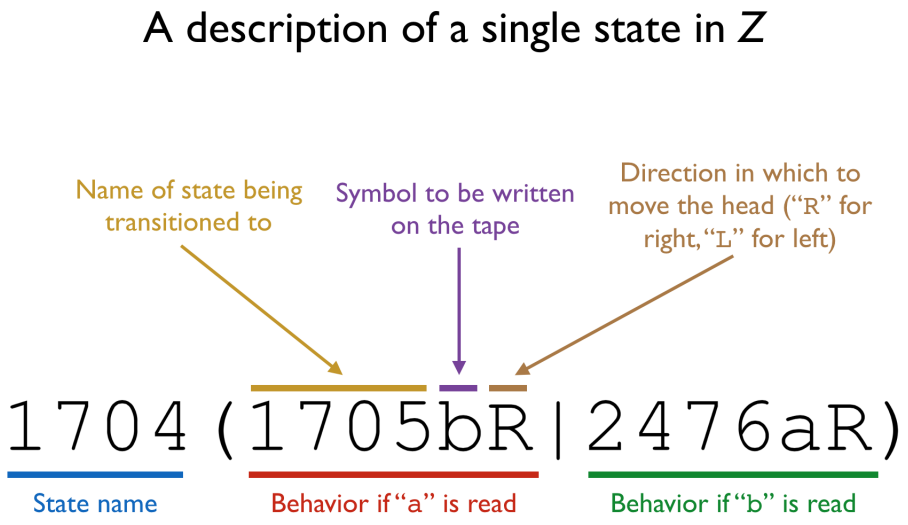


Figure 6: This figure explains how to read a description of a single state. Note that "`ERROR-`" or "`HALT--`" denote transitions to the `ERROR` or `HALT` states, respectively (no further information is provided because what symbol is written and which direction the head moves are at that point irrelevant).

We present below an explicit listing of Z. For a more easily readable version of Z, complete with descriptive state names, see [19].

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.

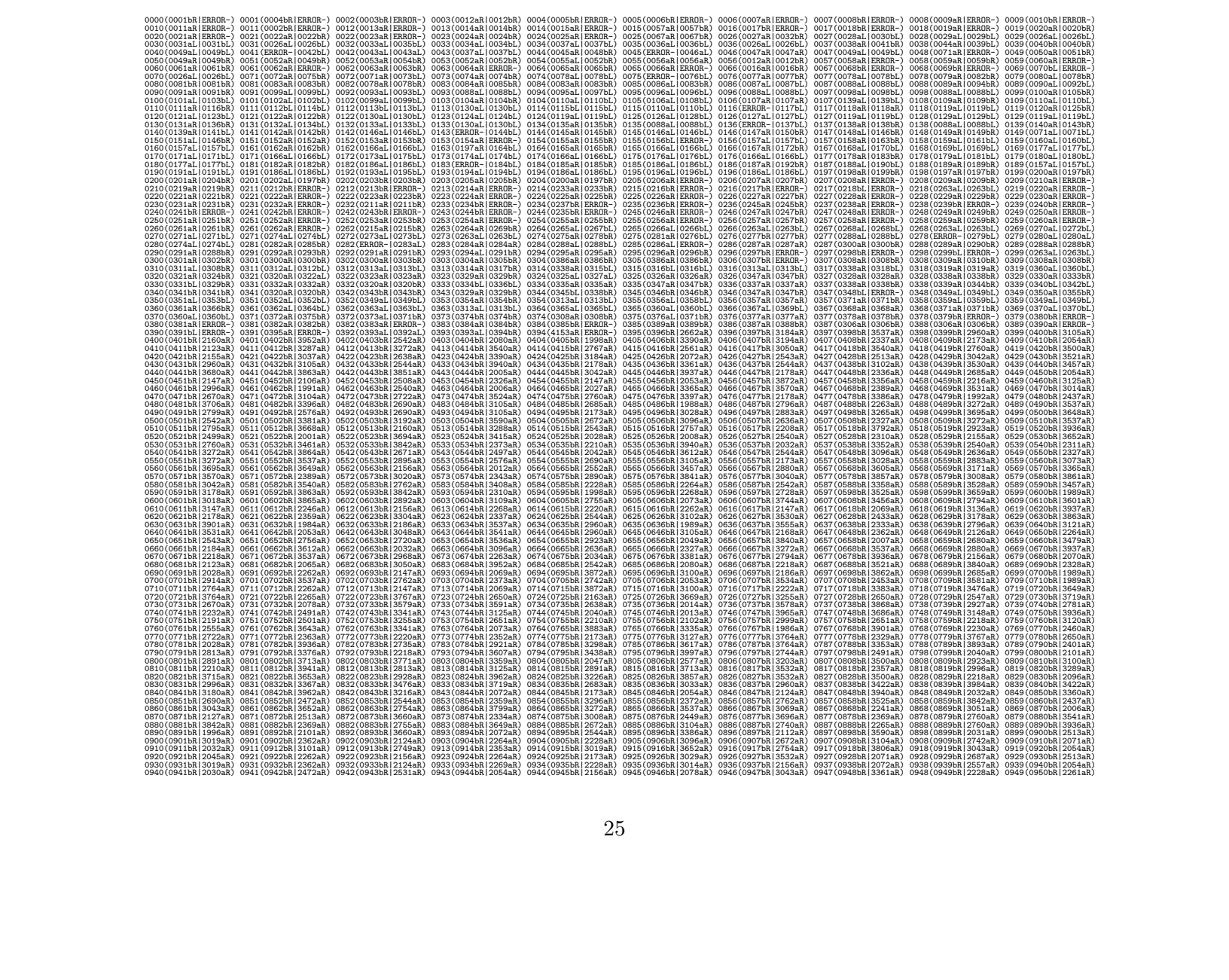
Figure 6 explains how to interpret the description shown below. In addition, note the following:

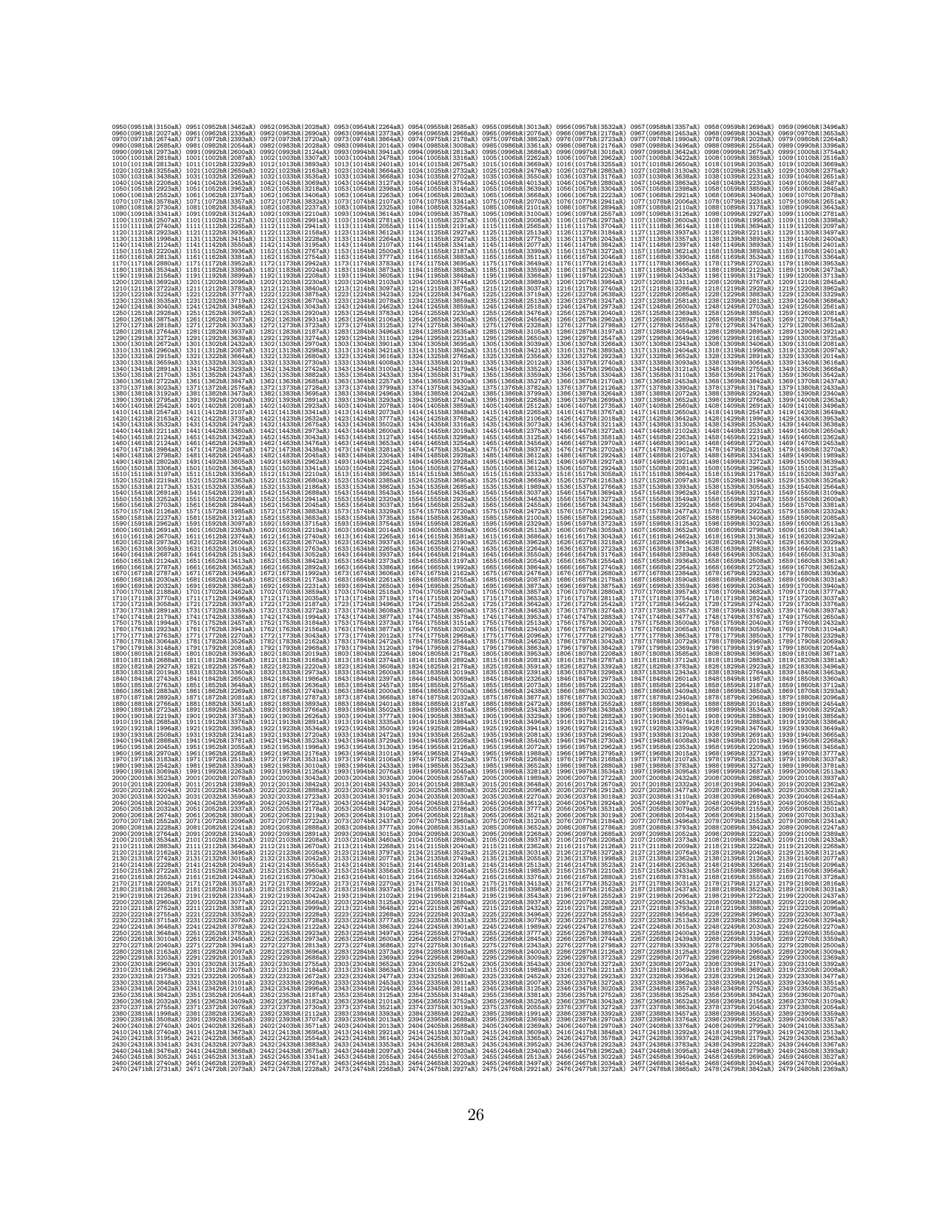
1. The tape has a 2-symbol alphabet, with tape symbols {a, b} and blank symbol a (in other words, a is the only symbol that can appear an infinite times on the tape).

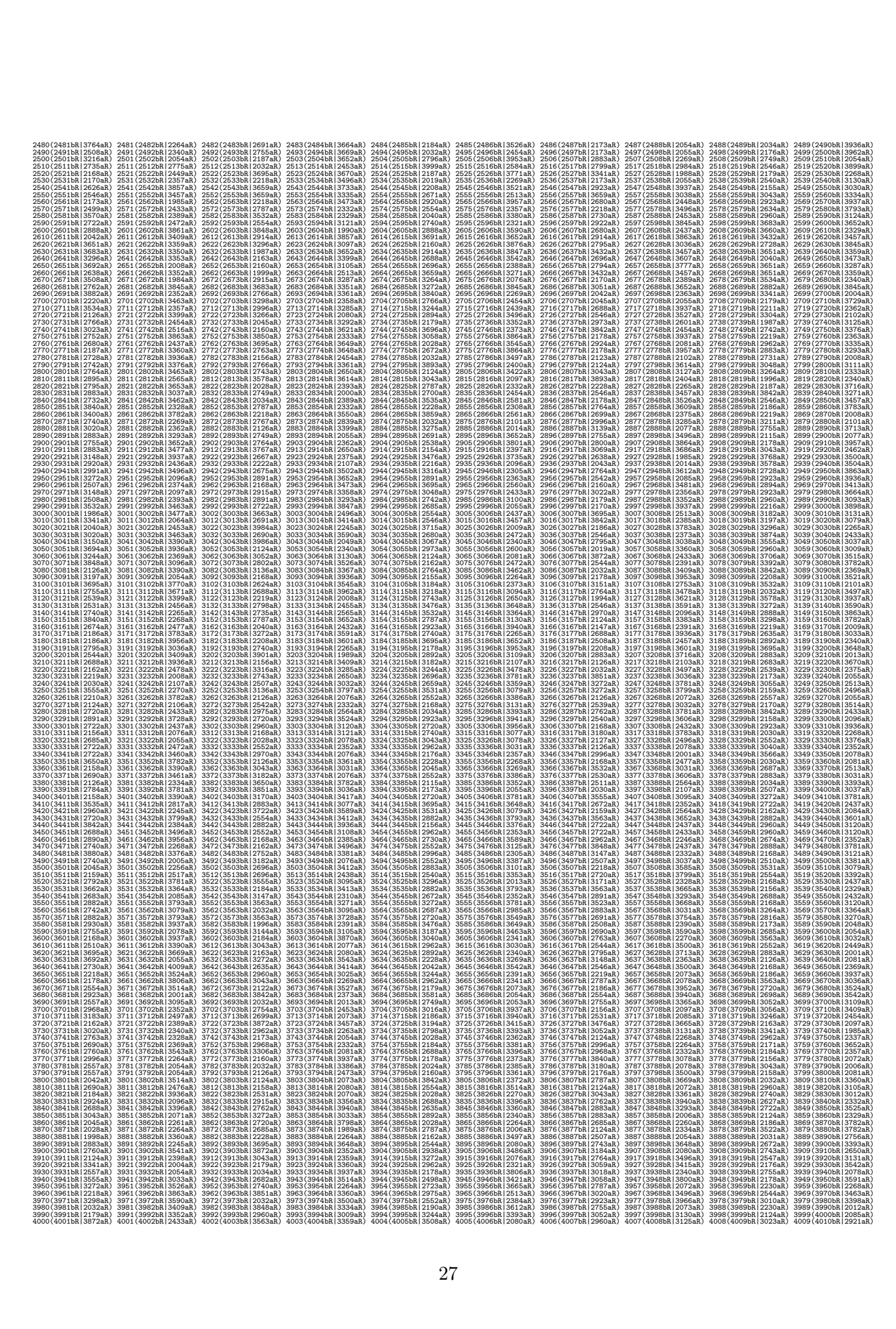
2. The start state of Z is 0000.

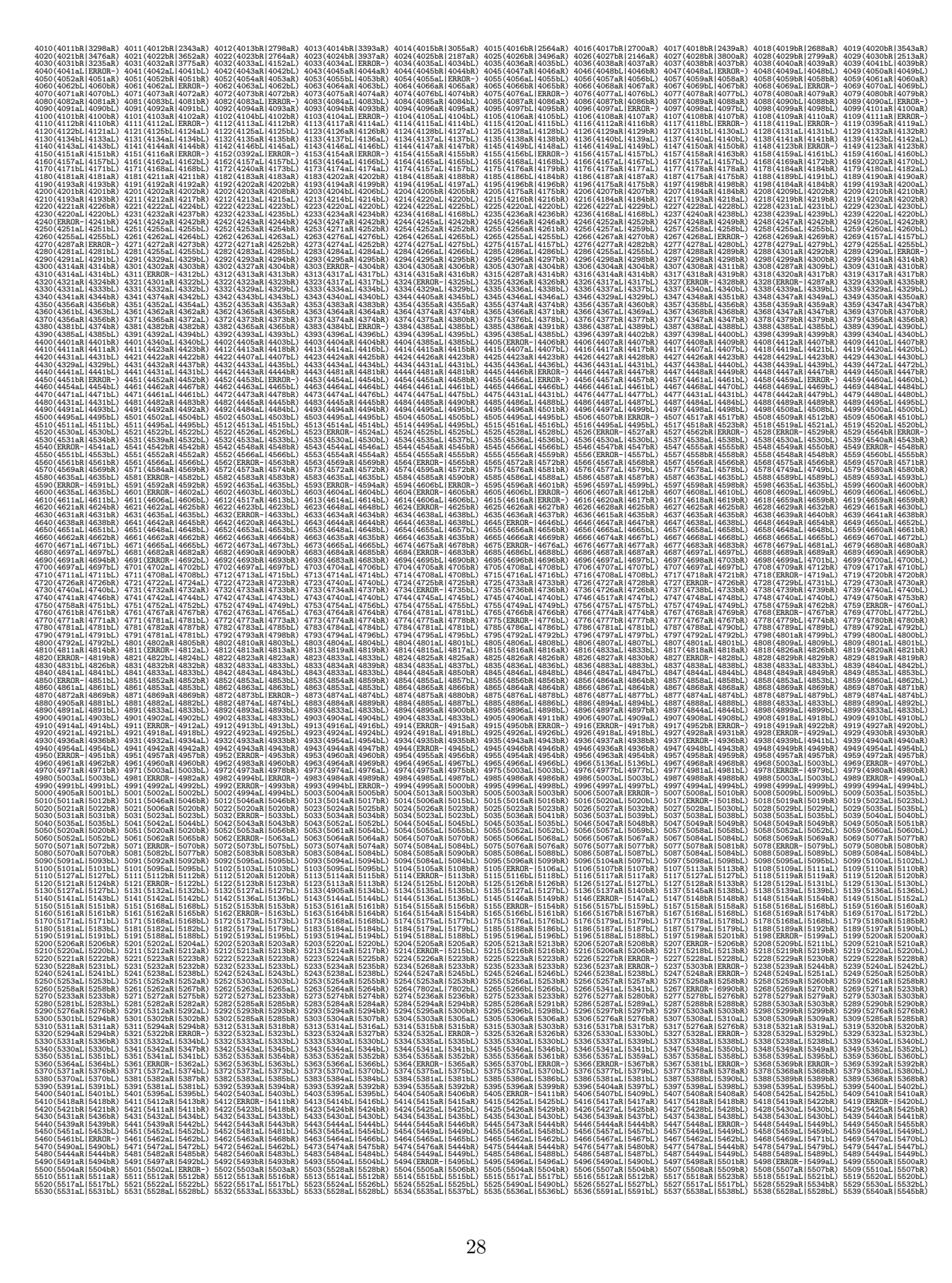
3. Z will never transition to the ERROR state. Any transition to the ERROR state could be replaced by a transition to any other state (including HALT) and the Turing machine's behavior would remain identical.

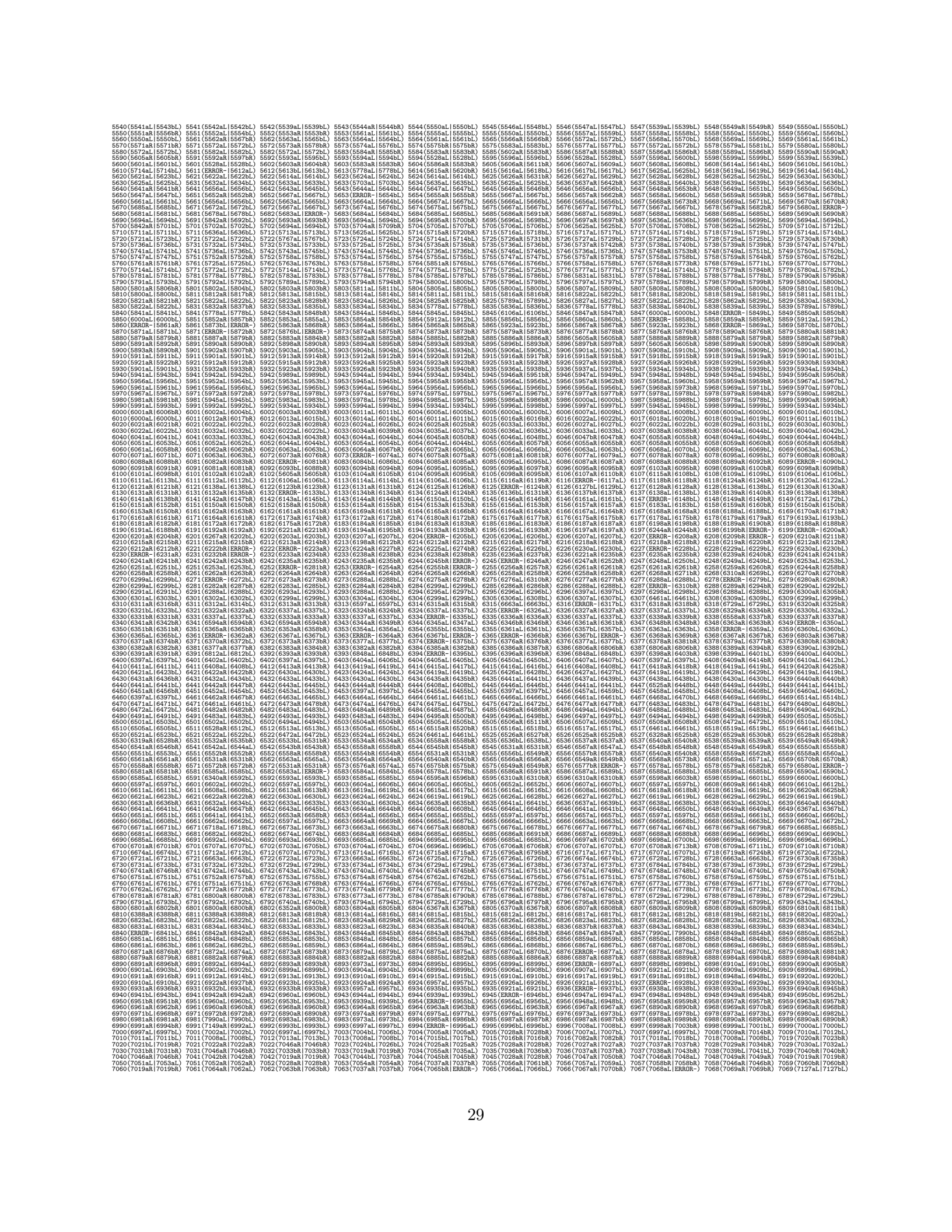
4. Z contains only one transition to the HALT state, out of state 7593.











7070(7071aL|ERROR-) 7071(7072aL|7072bL) 7072(7066aL|7066bL) 7073(7074aR|7077bR) 7074(7075bL|7073bR) 7075(7076aR|7076aR) 7076(7100aR|7100bR) 7077(7078bL|7080bL) 7078(7079aR|7079aR) 7079(7082aR|7082bR)
7080(7081aR|7081aR) 7081(7091aR|7091bR) 7082(7083aR|7088bR) 7083(7084aL|7086aL) 7084(7085bR|7085bR) 7085(7100aR|7100bR) 7086(7087bR|7087bR) 7087(7073aR|7073bR) 7088(7109aR|7089aL) 7089(7090bR|7090bR)
7090(7091aR|7091bR) 7091(7092aR|7097bR) 7092(7093bL|7095bL) 7093(7094bR|7094bR) 7094(7100aR|7100bR) 7095(7096bR|7096bR) 7096(7073aR|7073bR) 7097(7098bL|7091bR) 7098(7099bR|7099bR) 7099(7082aR|7082bR)
7100(7101aR|7104bR) 7101(7100aR|7102aL) 7102(7103aR|7103aR) 7103(7073aR|7073bR) 7104(7105aL|7107aL) 7105(7106aR|7106aR) 7106(7082aR|7082bR) 7107(7108aR|7108aR) 7108(7091aR|7091bR) 7109(7110aR|7115bR)
7110(7111aL|7113aL) 7111(7112bR|7112bR) 7112(7100aR|7100bR) 7113(7114bR|7114bR) 7114(7073aR|7073bR) 7115(7118aR|7116aL) 7116(7117bR|7117bR) 7117(7091aR|7091bR) 7118(7119bR|ERROR-) 7119(7120aL|7120bL)
7120(7121aR|7124bR) 7121(7122aL|ERROR-) 7122(7123aR|7123bR) 7123(7138aL|7138bL) 7124(7125aL|ERROR-) 7125(7126aL|7126bL) 7126(7120aL|7120bL) 7127(7128aR|7133bR) 7128(7129aL|7131bL) 7129(7130aL|7130bL)
7130(7127aL|7127bL) 7131(7132aL|7132bL) 7132(7127aL|7127bL) 7133(7134bL|7136bL) 7134(7135aR|7135aR) 7135(6990aR|6990bR) 7136(7137aL|7137bL) 7137(7127aL|7127bL) 7138(7139aR|7144bR) 7139(7140aL|7142bL)
7140(7141aL|7141bL) 7141(7138aL|7138bL) 7142(7143aL|7143bL) 7143(7138aL|7138bL) 7144(7145bL|7147bL) 7145(7146bR|7146bR) 7146(6990aR|6990bR) 7147(7148aL|7148bL) 7148(7138aL|7138bL) 7149(7150aR|7155bR)
7150(7151aL|7153aL) 7151(7152bL|7152bL) 7152(7222aL|7222bL) 7153(7154bL|7154bL) 7154(7158aL|7158bL) 7155(ERROR-|7156aL) 7156(7157bL|7157bL) 7157(7169aL|7169bL) 7158(7159aR|7164bR) 7159(7160aL|7162bL)
7160(7161aL|7161bL) 7161(7158aL|7158bL) 7162(7163aL|7163bL) 7163(7158aL|7158bL) 7164(7165aL|7167bL) 7165(7166aL|7166bL) 7166(7180aL|7180bL) 7167(7168aL|7168bL) 7168(7158aL|7158bL) 7169(7170aR|7175bR)
7170(7171aL|7173bL) 7171(7172aL|7172bL) 7172(7169aL|7169bL) 7173(7174aL|7174bL) 7174(7169aL|7169bL) 7175(7176aL|7178bL) 7176(7177aL|7177bL) 7177(7191aL|7191bL) 7178(7179aL|7179bL) 7179(7169aL|7169bL)
7180(7181aR|7186bR) 7181(7182aL|7184bL) 7182(7183aL|7183bL) 7183(7180aL|7180bL) 7184(7185aL|7185bL) 7185(7180aL|7180bL) 7186(7187bL|7189bL) 7187(7188aR|7188aR) 7188(7202aR|7202bR) 7189(7190aL|7190bL)
7190(7180aL|7180bL) 7191(7192aR|7197bR) 7192(7193aL|7195bL) 7193(7194aL|7194bL) 7194(7191aL|7191bL) 7195(7196aL|7196bL) 7196(7191aL|7191bL) 7197(7198bL|7200bL) 7198(7199bR|7199bR) 7199(7204aR|7204bR)
7200(7201aL|7201bL) 7201(7191aL|7191bL) 7202(7203bR|ERROR-) 7203(7206aR|7206bR) 7204(7205bR|ERROR-) 7205(7209aR|7209bR) 7206(7207aR|7208bR) 7207(7206aR|7206bR) 7208(7212aR|7206bR) 7209(7210aR|7211bR)
7210(7209aR|7209bR) 7211(7217aR|7209bR) 7212(7213aR|7214bR) 7213(7212aR|7212bR) 7214(7215bL|7212bR) 7215(7216aR|7216aR) 7216(7149aR|7149bR) 7217(7218aR|7219bR) 7218(7217aR|7217bR) 7219(7220bL|7217bR)
7220(7221bR|7221bR) 7221(7149aR|7149bR) 7222(7223aR|7228bR) 7223(7224aL|7226bL) 7224(7225aL|7225bL) 7225(7222aL|7222bL) 7226(7227aL|7227bL) 7227(7222aL|7222bL) 7228(7229aL|7231bL) 7229(7230aR|7230aR)
7230(7288aR|7288bR) 7231(7232aL|7232bL) 7232(7222aL|7222bL) 7233(7234aR|7239bR) 7234(7235aL|7237bL) 7235(7236aL|7236bL) 7236(7233aL|7233bL) 7237(7238aL|7238bL) 7238(7233aL|7233bL) 7239(7240aL|7242bL)
7240(7241aL|7241bL) 7241(7244aL|7244bL) 7242(7243aL|7243bL) 7243(7233aL|7233bL) 7244(7245aR|7250bR) 7245(7246aL|7248bL) 7246(7247aL|7247bL) 7247(7344aL|7344bL) 7248(7249aL|7249bL) 7249(7244aL|7244bL)
7250(7251aL|7253bL) 7251(7252aL|7252bL) 7252(7244aL|7244bL) 7253(7254aL|7254bL) 7254(7244aL|7244bL) 7255(7256aR|7261bR) 7256(7257aL|7259bL) 7257(7258aL|7258bL) 7258(7355aL|7355bL) 7259(7260aL|7260bL)
7260(7255aL|7255bL) 7261(7262aL|7264bL) 7262(7263aL|7263bL) 7263(7255aL|7255bL) 7264(7265aL|7265bL) 7265(7255aL|7255bL) 7266(7267bL|ERROR-) 7267(7268aL|7268bL) 7268(7269aR|7274bR) 7269(7270aL|7272bL)
7270(7271aL|7271bL) 7271(7268aL|7268bL) 7272(7273aL|7273bL) 7273(7268aL|7268bL) 7274(7275aL|7277bL) 7275(7276aL|7276bL) 7276(7364aL|7364bL) 7277(7278aL|7278bL) 7278(7268aL|7268bL) 7279(7280aR|7283bR)
7280(7281bL|7279bR) 7281(7282aR|7282aR) 7282(7306aR|7306bR) 7283(7284bL|7286bL) 7284(7285aR|7285aR) 7285(7288aR|7288bR) 7286(7287aR|7287aR) 7287(7297aR|7297bR) 7288(7289aR|7294bR) 7289(7290aL|7292aL)
7290(7291bR|7291bR) 7291(7306aR|7306bR) 7292(7293bR|7293bR) 7293(7279aR|7279bR) 7294(7315aR|7295aL) 7295(7296bR|7296bR) 7296(7297aR|7297bR) 7297(7298aR|7303bR) 7298(7299bL|7301bL) 7299(7300bR|7300bR)
7300(7306aR|7306bR) 7301(7302bR|7302bR) 7302(7279aR|7279bR) 7303(7304bL|7297bR) 7304(7305bR|7305bR) 7305(7288aR|7288bR) 7306(7307aR|7310bR) 7307(7306aR|7308aL) 7308(7309aR|7309aR) 7309(7279aR|7279bR)
7310(7311aL|7313aL) 7311(7312aR|7312aR) 7312(7288aR|7288bR) 7313(7314aR|7314aR) 7314(7297aR|7297bR) 7315(7316aR|7321bR) 7316(7317aL|7319aL) 7317(7318bR|7318bR) 7318(7306aR|7306bR) 7319(7320bR|7320bR)
7320(7279aR|7279bR) 7321(7324aR|7322aL) 7322(7323bR|7323bR) 7323(7297aR|7297bR) 7324(7325bR|ERROR-) 7325(7326aL|7326bL) 7326(7327aR|7330bR) 7327(7328aL|ERROR-) 7328(7329aR|7329bR) 7329(7333aL|7333bL)
7330(7331aL|ERROR-) 7331(7332aL|7332bL) 7332(7326aL|7326bL) 7333(7334aR|7339bR) 7334(7335aL|7337bL) 7335(7336aL|7336bL) 7336(7333aL|7333bL) 7337(7338aL|7338bL) 7338(7333aL|7333bL) 7339(7340aL|7342bL)
7340(7341aL|7341bL) 7341(7233aL|7233bL) 7342(7343aL|7343bL) 7343(7333aL|7333bL) 7344(7345aR|7350bR) 7345(7346aL|7348bL) 7346(7347aL|7347bL) 7347(7255aL|7255bL) 7348(7349aL|7349bL) 7349(7244aL|7244bL)
7350(7351aL|7353bL) 7351(7352aL|7352bL) 7352(7244aL|7244bL) 7353(7354aL|7354bL) 7354(7244aL|7244bL) 7355(7356aR|7359bR) 7356(7266aR|7357bL) 7357(7358aL|7358bL) 7358(7255aL|7255bL) 7359(7360aL|7362bL)
7360(7361aL|7361bL) 7361(7255aL|7255bL) 7362(7363aL|7363bL) 7363(7255aL|7255bL) 7364(7365aR|7368bR) 7365(7371aR|7366bL) 7366(7367bR|7367bR) 7367(7378aL|7378bL) 7368(ERROR-|7369bL) 7369(7370aL|7370aL)
7370(7364aL|7364bL) 7371(7372aR|7375bR) 7372(ERROR-|7373aL) 7373(7374aR|7374aR) 7374(7378aR|7378bR) 7375(7376aL|ERROR-) 7376(7377aR|7377aR) 7377(7402aR|7402bR) 7378(7379aR|7380bR) 7379(7378aR|7378bR)
7380(7381aR|7378bR) 7381(7382aR|7383bR) 7382(7381aR|7381bR) 7383(7384aL|7381bR) 7384(7385aR|7385aR) 7385(7386aR|7386bR) 7386(7387aR|7390bR) 7387(7388aL|7386bR) 7388(7389bL|7389bL) 7389(7391aL|7391bL)
7390(7386aR|7386bR) 7391(7392aR|7397bR) 7392(7393aL|7395bL) 7393(7394aL|7394bL) 7394(7391aL|7391bL) 7395(7396aL|7396bL) 7396(7391aL|7391bL) 7397(7398aL|7400bL) 7398(7399aL|7399bL) 7399(7364aL|7364bL)
7400(7401aL|7401bL) 7401(7391aL|7391bL) 7402(7403aR|7404bR) 7403(7402aR|7402bR) 7404(7405aL|7402bR) 7405(7406aR|7406aR) 7406(7407aR|7407bR) 7407(7408aR|7413bR) 7408(7409aL|7411aL) 7409(7410aR|7410bR)
7410(7600aL|7600bL) 7411(7412bR|7412bR) 7412(7416aR|7416bR) 7413(ERROR-|7414aL) 7414(7415bR|7415bR) 7415(7421aR|7421bR) 7416(7417aR|7418bR) 7417(7416aR|7416bR) 7418(7419bL|7416bR) 7419(7420aR|7420aR)
7420(7479aR|7479bR) 7421(7422aR|7423bR) 7422(7421aR|7421bR) 7423(7424bL|7421bR) 7424(7425bR|7425bR) 7425(7533aR|7533bR) 7426(7427aR|7432bR) 7427(7428aL|7430bL) 7428(7429aL|7429bL) 7429(7426aL|7426bL)
7430(7431aL|7431bL) 7431(7426aL|7426bL) 7432(7433aL|7435bL) 7433(7434aL|7434bL) 7434(7448aL|7448bL) 7435(7436aL|7436bL) 7436(7426aL|7426bL) 7437(7438aR|7443bR) 7438(7439aL|7441bL) 7439(7440aL|7440bL)
7440(7437aL|7437bL) 7441(7442aL|7442bL) 7442(7437aL|7437bL) 7443(7444aL|7446bL) 7444(7445aL|7445bL) 7445(7459aL|7459bL) 7446(7447aL|7447bL) 7447(7437aL|7437bL) 7448(7449aR|7454bR) 7449(7450aL|7452bL)
7450(7451aL|7451bL) 7451(7448aL|7448bL) 7452(7453aL|7453bL) 7453(7448aL|7448bL) 7454(7455bL|7457bL) 7455(7456aR|7456aR) 7456(7407aR|7407bR) 7457(7458aL|7458bL) 7458(7448aL|7448bL) 7459(7460aR|7465bR)
7460(7461aL|7463bL) 7461(7462aL|7462bL) 7462(7459aL|7459bL) 7463(7464aL|7464bL) 7464(7459aL|7459bL) 7465(7466bL|7468bL) 7466(7467bR|7467bR) 7467(7407aR|7407bR) 7468(7469aL|7469bL) 7469(7459aL|7459bL)
7470(7471aR|7474bR) 7471(7472bL|7470bR) 7472(7473aR|7473aR) 7473(7497aR|7497bR) 7474(7475bL|7477bL) 7475(7476aR|7476aR) 7476(7479aR|7479bR) 7477(7478aR|7478aR) 7478(7488aR|7488bR) 7479(7480aR|7485bR)
7480(7481aL|7483aL) 7481(7482bR|7482bR) 7482(7497aR|7497bR) 7483(7484bR|7484bR) 7484(7470aR|7470bR) 7485(7506aR|7486aL) 7486(7487bR|7487bR) 7487(7488aR|7488bR) 7488(7489aR|7494bR) 7489(7490bL|7492bL)
7490(7491bR|7491bR) 7491(7497aR|7497bR) 7492(7493bR|7493bR) 7493(7470aR|7470bR) 7494(7495bL|7488bR) 7495(7496bR|7496bR) 7496(7479aR|7479bR) 7497(7498aR|7501bR) 7498(7497aR|7499aL) 7499(7500aR|7500aR)
7500(7470aR|7470bR) 7501(7502aL|7504aL) 7502(7503aR|7503aR) 7503(7479aR|7479bR) 7504(7505aR|7505aR) 7505(7488aR|7488bR) 7506(7507aR|7512bR) 7507(7508aL|7510aL) 7508(7509bR|7509bR) 7509(7497aR|7497bR)
7510(7511bR|7511bR) 7511(7470aR|7470bR) 7512(7515aR|7513aL) 7513(7514bR|7514bR) 7514(7488aR|7489bR) 7515(7516bR|ERROR-) 7516(7517aL|7517bL) 7517(7518aR|7521bR) 7518(7519aL|ERROR-) 7519(7520aR|7520bR)
7520(7578aL|7578bL) 7521(7522aL|ERROR-) 7522(7523aL|7523bL) 7523(7517aL|7517bL) 7524(7525aR|7528bR) 7525(7526bL|7524bR) 7526(7527aR|7527aR) 7527(7551aR|7551bR) 7528(7529bL|7531bL) 7529(7530aR|7530aR)
7530(7533aR|7533bR) 7531(7532aR|7532aR) 7532(7542aR|7542bR) 7533(7534aR|7539bR) 7534(7535aL|7537aL) 7535(7536bR|7536bR) 7536(7551aR|7551bR) 7537(7538bR|7538bR) 7538(7524aR|7524bR) 7539(7560aR|7540aL)
7540(7541bR|7541bR) 7541(7542aR|7542bR) 7542(7543aR|7548bR) 7543(7544bL|7546bL) 7544(7545bR|7545bR) 7545(7551aR|7551bR) 7546(7547bR|7547bR) 7547(7524aR|7524bR) 7548(7549bL|7542bR) 7549(7550bR|7550bR)
7550(7533aR|7533bR) 7551(7552aR|7555bR) 7552(7551aR|7553aL) 7553(7554aR|7554aR) 7554(7524aR|7524bR) 7555(7556aL|7558aL) 7556(7557aR|7557aR) 7557(7533aR|7533bR) 7558(7559aR|7559aR) 7559(7542aR|7542bR)
7560(7561aR|7566bR) 7561(7562aL|7564aL) 7562(7563bR|7563bR) 7563(7551aR|7551bR) 7564(7565bR|7565bR) 7565(7524aR|7524bR) 7566(7569aR|7567aL) 7567(7568bR|7568bR) 7568(7542aR|7542bR) 7569(7570bR|ERROR-)
7570(7571aL|7571bL) 7571(7572aR|7575bR) 7572(7573aL|ERROR-) 7573(7574aR|7574bR) 7574(7589aL|7589bL) 7575(7576aL|ERROR-) 7576(7577aL|7577bL) 7577(7571aL|7571bL) 7578(7579aR|7584bR) 7579(7580aL|7582bL)
7580(7581aL|7581bL) 7581(7578aL|7578bL) 7582(7583aL|7583bL) 7583(7578aL|7578bL) 7584(7585aL|7587bL) 7585(7586aL|7586bL) 7586(7426aL|7426bL) 7587(7588aL|7588bL) 7588(7578aL|7578bL) 7589(7590aR|7595bR)
7590(7591aL|7593bL) 7591(7592aL|7592bL) 7592(7589aL|7589bL) 7593(7594aL|7594bL) 7594(7589aL|7589bL) 7595(7596aL|7598bL) 7596(7597aL|7597bL) 7597(7437aL|7437bL) 7598(7599aL|7599bL) 7599(7589aL|7589bL)
7600(7601aR|7606bR) 7601(7602aL|7604bL) 7602(7603aL|7603bL) 7603(7600aL|7600bL) 7604(7605aL|7605bL) 7605(7600aL|7600bL) 7606(7609aR|7607bL) 7607(7608aL|7608bL) 7608(7600aL|7600bL) 7609(7610aR|7613bR)
7610(7609aR|7611bL) 7611(7612aR|7612bR) 7612(7616aL|7616bL) 7613(ERROR-|7614bL) 7614(7615aR|7615bR) 7615(7616aL|7616bL) 7616(7617aR|7618bR) 7617(7619aR|7616bR) 7618(7616aR|7616bR) 7619(7620bR|ERROR-)
7620(7621aR|7621bR) 7621(7622aR|7623bR) 7622(7621aR|7621bR) 7623(7624aR|7621bR) 7624(7625aR|7626bR) 7625(7624aR|7624bR) 7626(7627aL|7624bR) 7627(7628aR|7628aR) 7628(7629aR|7629bR) 7629(7630aR|7635bR)
7630(7631aL|7633bL) 7631(7632aR|7632bR) 7632(7638aL|7638bL) 7633(7634aR|7634bR) 7634(7149aL|7149bL) 7635(ERROR-|7636bL) 7636(7637aR|7637bR) 7637(7149aL|7149bL) 7638(7639aR|7644bR) 7639(7640aL|7642bL)
7640(7641aL|7641bL) 7641(7638aL|7638bL) 7642(7643aL|7643bL) 7643(7638aL|7638bL) 7644(7645aL|7647bL) 7645(7646aL|7646bL) 7646(7649aL|7649bL) 7647(7648aL|7648bL) 7648(7638aL|7638bL) 7649(7650aR|7655bR)
7650(7651aL|7653bL) 7651(7652aL|7652bL) 7652(7649aL|7649bL) 7653(7654aL|7654bL) 7654(7649aL|7649bL) 7655(7656aL|7658bL) 7656(7657aR|7657aR) 7657(7660aL|7660bL) 7658(7659aL|7659bL) 7659(7649aL|7649bL)
7660(7661aR|7666bR) 7661(7662aL|7664bL) 7662(7663aL|7663bL) 7663(7660aL|7660bL) 7664(7665aR|7665bR) 7665(7669aL|7669bL) 7666(ERROR-|7667bL) 7667(7668aR|7668bR) 7668(7669aL|7669bL) 7669(7670aR|7675bR)
7670(7671aL|7673bL) 7671(7672bR|7672bR) 7672(7680aL|7680bL) 7673(7674aL|7674bL) 7674(7669aL|7669bL) 7675(7676aL|7678bL) 7676(7677aL|7677bL) 7677(7669aL|7669bL) 7678(7679aL|7679bL) 7679(7669aL|7669bL)
7680(7681aR|7686bR) 7681(7682aL|7684aL) 7682(7683aL|7683bL) 7683(7721aL|7721bL) 7684(7685bL|7685bL) 7685(7689aL|7689bL) 7686(7687aR|7687aL) 7687(7688bL|7688bL) 7688(7691aL|7691bL) 7689(ERROR-|7690aR)
7690(7693bR|ERROR-) 7691(ERROR-|7692bR) 7692(7695bR|ERROR-) 7693(ERROR-|7694bR) 7694(7702aR|7702bR) 7695(ERROR-|7696bR) 7696(7707aR|7707bR) 7697(7698aR|7699bR) 7698(7697aR|7697bR) 7699(7700aL|7697bR)
7700(7701aR|7701aR) 7701(7746aR|7746bR) 7702(7703aR|7704bR) 7703(7702aR|7702bR) 7704(7705bL|7702bR) 7705(7706aR|7706aR) 7706(7746aR|7746bR) 7707(7708aR|7709bR) 7708(7707aR|7707bR) 7709(7710bL|7707bR)
7710(7711bR|7711bR) 7711(7746aR|7746bR) 7712(7713aR|7718bR) 7713(7714aL|7716bL) 7714(7715aL|7715bL) 7715(7712aL|7712bL) 7716(7717aL|7717bL) 7717(7712aL|7712bL) 7718(7680aR|7719bL) 7719(7720aL|7720bL)
7720(7712aL|7712bL) 7721(ERROR-|7722bR) 7722(7723bL|ERROR-) 7723(7724aL|7724bL) 7724(7725aL|7725bL) 7725(7726aL|7721bR) 7726(7727aL|7728bR) 7727(7728aR|7728bR) 7728(7734aL|7734bL) 7729(7730aL|7730aL)
7730(7725aL|7725bL) 7731(ERROR-|7732aL) 7732(7733aL|7733aL) 7733(7725aL|7725bL) 7734(7735aR|7736bR) 7735(7734aR|7734bR) 7736(7734aR|7990bR) 7737(7738aR|7741bR) 7738(7739bL|7737bR) 7739(7740aR|7740aR)
7740(7764aR|7764bR) 7741(7742bL|7744bL) 7742(7743aR|7743aR) 7743(7746aR|7746bR) 7744(7745aR|7745aR) 7745(7755aR|7755bR) 7746(7747aR|7752bR) 7747(7748aL|7750aL) 7748(7749bR|7749bR) 7749(7764aR|7764bR)
7750(7751bR|7751bR) 7751(7737aR|7737bR) 7752(7773aR|7753aL) 7753(7754bR|7754bR) 7754(7755aR|7755bR) 7755(7756aR|7761bR) 7756(7757bL|7759bL) 7757(7758bR|7758bR) 7758(7764aR|7764bR) 7759(7760bR|7760bR)
7760(7737aR|7737bR) 7761(7762bL|7764aL) 7762(7763bR|7763bR) 7763(7746aR|7746bR) 7764(7765aR|7768bR) 7765(7764aR|7766aL) 7766(7767aR|7767aR) 7767(7737aR|7737bR) 7768(7769aL|7771aL) 7769(7770aR|7770aR)
7770(7746aR|7746bR) 7771(7772aR|7772aR) 7772(7755bR|7773aL) 7773(7774aR|7779bR) 7774(7775aL|7777aL) 7775(7776bR|7776bR) 7776(7764aR|7764bR) 7777(7778bR|7778bR) 7778(7737aR|7737bR) 7779(7782aR|7780aL)
7780(7781bR|7781bR) 7781(7755aR|7755bR) 7782(7783bR|ERROR-) 7783(7784aL|7784bL) 7784(7785aR|7788bR) 7785(7786aL|ERROR-) 7786(7787aR|7787bR) 7787(7791aL|7791bL) 7788(7789aL|ERROR-) 7789(7790aL|7790bL)
7790(7784aL|7784bL) 7791(7792aR|7797bR) 7792(7793aL|7795bL) 7793(7794aL|7794bL) 7794(7791aL|7791bL) 7795(7796aL|7796bL) 7796(7791aL|7791bL) 7797(7798aL|7800bL) 7798(7799aL|7799bL) 7799(7712aL|7712bL)
7800(7801aL|7801bL) 7801(7794aL|7794bL) 7802(7803aR|7808bR) 7803(7804aL|7806bL) 7804(7805aL|7805bL) 7805(7802aL|7802bL) 7806(7807aL|7807bL) 7807(7802aL|7802bL) 7808(7809aL|7811bL) 7809(7810aL|7810aL)
7810(7813aL|7813bL) 7811(7812aL|7812bL) 7812(7802aL|7802bL) 7813(7814aR|7819bR) 7814(7815aL|7817bL) 7815(7816aL|7816bL) 7816(7813aL|7813bL) 7817(7818aR|7818bR) 7818(7822aL|7822bL) 7819(ERROR-|7820bL)
7820(7821aR|7821bR) 7821(7822aL|7822bL) 7822(7823aR|7828bR) 7823(7824aL|7826bL) 7824(7825bL|7825bL) 7825(7833aL|7833bL) 7826(7827aL|7827bL) 7827(7822aL|7822bL) 7828(7829aL|7831bL) 7829(7830aL|7830bL)
7830(7822aL|7822bL) 7831(7832aL|7832bL) 7832(7822aL|7822bL) 7833(7834aR|7839bR) 7834(7835aL|7837bL) 7835(7836aL|7836bL) 7836(7833aL|7833bL) 7837(7838aL|7838bL) 7838(7833aL|7833bL) 7839(7840aL|7842bL)
7840(7841aL|7841bL) 7841(7842aL|7842bL) 7842(7843aR|7843bL) 7843(7844aL|7833bL) 7844(7845aL|ERROR-) 7845(7846aL|7846bL) 7846(7847aL|7852bR) 7847(7848aL|7849bL) 7848(7849aR|7849bR) 7849(7850aL|7850bL)
7850(7851aL|7851aL) 7851(7846aL|7846bL) 7852(ERROR-|7853aL) 7853(7854aL|7854aL) 7854(7846aL|7846bL) 7855(7856aR|7857bR) 7856(7855aR|7855bR) 7857(7858aR|7855bR) 7858(7859aR|7860bR) 7859(7858aR|7858bR)
7860(7861aL|7858bR) 7861(7862aR|7862aR) 7862(7863aR|7863bR) 7863(7864aR|7869bR) 7864(7865aL|7867aL) 7865(7866aR|7866bR) 7866(7946aL|7946bL) 7867(7868bL|7868bL) 7868(7872aL|7872bL) 7869(ERROR-|7870aL)
7870(7871bL|7871bL) 7871(7883aL|7883bL) 7872(7873aR|7878bR) 7873(7874aL|7876bL) 7874(7875aL|7875bL) 7875(7872aL|7872bL) 7876(7877aL|7877bL) 7877(7872aL|7872bL) 7878(7879aL|7881bL) 7879(7880aL|7880bL)
7880(7894aL|7894bL) 7881(7882aL|7882bL) 7882(7872aL|7872bL) 7883(7884aR|7889bR) 7884(7885aL|7887bL) 7885(7886aL|7886bL) 7886(7883aL|7883bL) 7887(7888aL|7888bL) 7888(7883aL|7883bL) 7889(7890aL|7892bL)
7890(7891aL|7891bL) 7891(7921aL|7921bL) 7892(7893aL|7893bL) 7893(7883aL|7883bL) 7894(7895aR|7900bR) 7895(7896bL|7898bL) 7896(7897aR|7897aR) 7897(7930aL|7930bL) 7898(7899aL|7899bL) 7899(7894aL|7894bL)
7900(ERROR-|7901bL) 7901(7902aL|7902aL) 7902(7903aL|7903bL) 7903(7904aR|7909bR) 7904(7905bL|7907bL) 7905(7906bR|7906bR) 7906(7930aL|7930bL) 7907(7908bL|7908bL) 7908(7894aL|7894bL) 7909(ERROR-|7910bL)
7910(7911aL|7911bL) 7911(7903aL|7903bL) 7912(7913aR|7918bR) 7913(7914bL|7916bL) 7914(7915aR|7915aR) 7915(7933aL|7933bL) 7916(7917aL|7917bL) 7917(7912aL|7912bL) 7918(ERROR-|7919bL) 7919(7920aL|7920aL)
7920(7921aL|7921bL) 7921(7922aR|7927bR) 7922(7923bL|7925bL) 7923(7924bR|7924bR) 7924(7933aL|7933bL) 7925(7926bL|7926bL) 7926(7912aL|7912bL) 7927(ERROR-|7928bL) 7928(7929aL|7929bL) 7929(7921aL|7921bL)
7930(7931aR|7932bR) 7931(7930aR|7930bR) 7932(7936aR|7935bR) 7933(7934aR|7935bR) 7934(7933aR|7933bR) 7935(7941aR|7933bR) 7936(7937aR|7938bR) 7937(7936aR|7936bR) 7938(7939bL|7936bR) 7939(7940aR|7940aR)
7940(7863aR|7863bR) 7941(7942aR|7943bR) 7942(7941aR|7941bR) 7943(7944bL|7941bR) 7944(7945bR|7945bR) 7945(7863aR|7863bR) 7946(7947aR|7952bR) 7947(7948aL|7950aL) 7948(7949aL|7949bL) 7949(7955aL|7955bL)
7950(7951aL|7951aL) 7951(7946aL|7946bL) 7952(ERROR-|7953aL) 7953(7954aL|7954aL) 7954(7946aL|7946bL) 7955(7956aR|7961bR) 7956(7957aL|7959aL) 7957(7958aL|7958bL) 7958(7964aL|7964bL) 7959(7960aL|7960aL)
7960(7946aL|7946bL) 7961(ERROR-|7962aL) 7962(7963aL|7963aL) 7963(7946aL|7946bL) 7964(7965aR|7966bR) 7965(HALT--|7967bR) 7966(ERROR-|7967bR) 7967(7968aR|7973bR) 7968(7969aL|7971bL) 7969(7970aL|7970bL)
7970(7967aL|7967bL) 7971(7972aL|7972bL) 7972(7967aL|7967bL) 7973(7974aL|7976bL) 7974(7975aL|7975aL) 7975(7978aL|7978bL) 7976(7977aL|7977bL) 7977(7967aL|7967bL) 7978(7979aR|7984bR) 7979(7980bL|7982bL)
7980(7981bR|7981bR) 7981(7987aL|7987bL) 7982(7983bL|7983bL) 7983(7978aL|7978bL) 7984(ERROR-|7985bL) 7985(7986aR|7986aR) 7986(7987aL|7987bL) 7987(7988aR|7989bR) 7988(7990aR|7987bR) 7989(7987aR|7987bR)
7990(7991aR|7992bR) 7991(7990aR|ERROR-) 7992(ERROR-|7993bL) 7993(7994aR|7994bR) 7994(7995aL|7995bL) 7995(7996aR|7997bR) 7996(7998aR|7995bR) 7997(7995aR|7995bR) 7998(7999aR|8000bR) 7999(8001aR|7995bR)
8000(7995aR|7995bR) 8001(8002aR|8005bR) 8002(8003aL|7995bR) 8003(8004aR|8004bR) 8004(8006aL|8006bL) 8005(7995aR|7995bR) 8006(8007aR|8008bR) 8007(8006aR|ERROR-) 8008(ERROR-|8009bL) 8009(8010aL|8010bL)
8010(8011aL|8011bL) 8011(8012bL|ERROR-) 8012(4484aL|4484bL)