Executing Brainfuck on a Single-Taped Turing Machine

An Honors Thesis for the Department of Computer Science

Daniel Scherzer

Tufts University, 2024

Contents

1 Introduction

Turing Machines were first introduced in Alan Turing's seminal 1936 paper "On Computable Numbers, with an Application to the Entscheidungsproblem", though he referred to them as "computing machines" (Turing 1936). A Turing machine is a model for an abstract computation machine, and the ability to simulate the computation of a Turing machine is referred to as "Turing-completeness". In addition to the Turing machine itself, many processes that do not at first glance appear to be computers have nevertheless been proven to be Turing-complete. These include the card game *Magic: The Gathering* (Churchill, Biderman, and Herrick 2019), the "Rule 110" cellular automaton (Cook 2004), Conway's "Game of Life" (Rendell 2014), the `printf` print function in C (Carlini et al. 2015), and even the human heart (Scarle 2009; Ostrovsky 2009).

Various programming languages have also been shown to be Turing-complete, or effectively so.[1] The Church–Turing thesis argues that the converse is also true, that a simple Turing Machine is able to compute anything that these languages can (Sipser 2013, 165, 281). But how? The Church–Turing thesis doesn't explicitly show how to implement any given programming language on a Turing Machine. My goal is to demonstrate, in an understandable manner, exactly how a Turing Machine can be configured to do the same things that a modern programming language can do. I do this for the idea of the abstract Turing Machine, but since (for example) Conway's "Game of Life" is able to simulate such a machine, it should also be able to execute the program that I write for such a machine, allowing the execution of code in that game.

A key question, then, is what programming language to try and support. I decided to go with the language "Brainfuck", which is Turing-complete but relatively easy to implement because of its extremely limited syntax. It is that limited syntax that causes it to often be referred to as a "Turing tarpit," since the limitations make it hard to write useful code in the language (Chandra 2014, 119). But, Brainfuck serves as a relatively simple first step for identifying how to support other programming languages.

1 (a) Outline

The rest of section one deals with general notes about the way that Turing Machines can be defined and how they will be implemented. Section two deals with how to provide shortcuts to a developer programming single-taped Turing Machines. It proves that the shortcuts provide no new computational func-

[1] Since Turing Machines, and thus other systems that are Turing-complete, technically requires infinite memory, implementations of such languages are unlikely to be Turing-complete (Sipser 2013, 166). But, the specification of the language can be. For some languages, however, the *specification* may not even be Turing-complete (Poss 2018). Proving the Turing-completeness or incompleteness of various programming languages is outside of the scope of this thesis.

tionality by explaining how to implement the same features on a machine that lacks the shortcuts. Section three deals with switching to multi-taped Turing Machines, and again proves that these provide no new computational functionality by explaining how to simulate a multi-taped Turing Machine using only a single-taped Turing Machine. Section four describes how to actually implement Brainfuck on a multi-taped Turing Machine while making use of various shortcuts, which again do not provide any new computational functionality. Finally, section five concludes with a discussion on potential future work.

Eight appendices are included at the end. The first two list the constants and symbols (first appendix) and generated state names (second appendix) that the developer and compiler make use of. The third appendix contains a step-by-step walk-through of one of the more complicated programs from section two of the paper. The fourth appendix lists the actual states and transitions needed to simulate a multi-taped Turing Machine for a specific program explained in section three. The fifth appendix describes the relationship between Brainfuck and a prior language, P''. Appendices six and seven deal with Brainfuck, providing operational semantics for the version that I implement and then the actual implementation. Finally, the last appendix includes a high-level overview of the code that I wrote for the compiler.

That compiler can be tried out for a limited time on my Tufts EECS website at `https://www.eecs.tufts.edu/~dscher02/Thesis/Demo.html`. The code for the compiler is also attached to this thesis and can be run locally.

To be clear, this document is meant to be viewed as a specification for a Turing Machine compiler that provides extra features for developers, even if those features are not available on the underlying machine that the program gets run on. The attached code is an implementation of this specification.

1 (b) Terms

Let us begin by defining some terms to reduce confusion.

- TM: an initialism for "Turing Machine"
- Developer: the **developer** seeks to program a TM in a manner that may make use of extra features that the TM may not possess, but which the compiler will ensure are available.
- Compiler: the **compiler** is the logic (and eventually code) that converts the program from the developer into one that can be understood by the underlying machine. For any features that the developer uses that the machine does not natively support, the compiler will translate the feature usage into configuration that *is* supported by the machine.[2]

[2]In places, the compiler acts more akin to an interpreter, since it includes the logic for the underlying machine, or more akin to a transpiler, when it fills in functionality that is

- Program: the **program** refers to the set of states, tape alphabet, transition function, and later the number of tapes of the definition of the Turing Machine. To support various features, the compiler may add additional states and alphabet symbols and manipulate the transition function to allow the developer's program to use shortcuts that the underlying machine does not support.

- Machine: the **machine** of the underlying TM is the platform on which the developer's states and transitions will be run. It may or may not natively support any of the extra features described in this document, but the developer is free to assume that all of the extra features are supported, with the compiler providing them if needed.

- Index: the location of the "head" of a TM tape, where "`0`" refers to the leftmost cell on the tape, "`1`" refers to the cell directly to the right of that leftmost cell, and subsequent indices likewise indicate the number of cells from the start (left) of the tape the head is located.

Turing Machines can be defined in various ways. The formal definition given by Sipser in his book "Introduction to the Theory of Computation" is what I will use, where a Turing Machine is defined with the following seven parts (Sipser 2013, 168, Def 3.3).

- Q - the set of states
- Σ - the input alphabet (which must not include the blank symbol "$\sqcup$")
- Γ - the tape alphabet, which must include the input alphabet and the empty symbol[3]
- δ - a transition function in the form $Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\}$
- q_0 - a start state in Q
- q_{accept} - an accept state in Q
- q_{reject} - a reject state in Q ($q_{accept} \neq q_{reject}$)

Note that I am not going to go into depth explaining how a basic Turing Machine works. Sipser provides a great introduction for those unfamiliar with their operation, and there are plenty of resources available online on the topic. I will note, however, that there are multiple (computationally equivalent) models of Turing Machines. I assume that the underlying machine is of the model that requires every transition to both write a symbol and move the tape head either left or right, and that moving the head of the tape left when it is already at the start of the tape does not change the position of the tape head (Sipser 2013, 165-6).

---

not available in the underlying machine. "Compiler" seemed to be the best-fitting label to use, but it performs multiple functions.

[3]The terms "empty symbol" and "blank symbol" are used interchangeably to refer to the symbol indicating that a tape cell has no actual content in it, which is represented with "$\sqcup$".

To avoid repeating a lot of boilerplate, let the start state (q_0), accept state (q_{accept}), and reject state (q_{reject}) always be "`START`", "`ACCEPT`", and "`REJECT`" respectively.

Various example programs are used in the process of explaining how to make Turing Machines easier to use. Programs are displayed in the following format:

Program #123: Name
Short description
Configuration of states and transitions explained with prose and grouped by target state
Equivalent pseudocode for the configuration of states and transitions.

Within the pseudocode, various constants are available for use. These include the names of the accept and reject states, shortcuts to specify movements and targeted symbols, and more, and are added as the features are introduced. An overall list of the code constants is available in an appendix; note that not all of the constants listed there are available for use with a normal Turing Machine; some are created as features are added over the course of this explanation.

1 (c) The transition function

We can only assume that the machine supports transitions in the form

$$Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\}$$

but we want to allow the developer to write transitions in other forms, and the compiler will expand these as needed. Additionally, we want to make it easier for the developer to only specify transitions that they need, and omit the rest. Accordingly, instead of providing the compiler with an actual function, the developer builds up the transition function by defining a series of tuples, one for each transition (at least to begin with; by the end a single tuple can be used define many transitions at once). As the compiler provides additional functionality, the developer will have more ways to define transitions, but to begin with the tuples are in the form:

$$Q \times \Gamma \times Q \times \Gamma \times \{L, R\}$$

which means that when the state is the value in Q given as the first element in the tuple, and the symbol at the head of the tape is the value in Γ given as the second element of the tuple, the compiler should return the third, fourth, and fifth elements of the tuple as the result of the transition function.

To allow explaining the effect of other forms of transition definitions more precisely later, let us consider this tuple to be the inputs to some abstract function DefTransition, such that DefTransition$(q_f, \gamma_f, q_t, \gamma_t, d)$ creates a new transition from the state q_f and symbol γ_f to the state q_t while writing the symbol γ_t and then moving in direction d.

The developer **must not** provide multiple tuples that begin with the same $Q \times \Gamma$ values but have different transition results, since the transition function must be deterministic.[4] For any transitions that are not defined by the developer, the compiler will fill in the resulting value as $\{\text{REJECT} \times \text{"}\sqcup\text{"} \times R\}$ (write the empty symbol, move right, and reject). Such a transition should always be valid, since REJECT will be the reject state and thus a valid state, and "$\sqcup$" is always a valid symbol on the tape.

Since the results of a transition are not validated when the transition is defined (for example, to allow defining the states of a machine in any desired order), the produced transition function will also require that the inputs are valid states and symbols. On a lower level, the compiler will produce a transition function that works essentially as follows:

1. Receives a tuple in the form $\{q_s \times \gamma_s\}$, expecting that $q_s \in Q$ and $\gamma_s \in \Gamma$.
2. If $q_s \notin Q$, return $\{\text{REJECT} \times \text{"}\sqcup\text{"} \times R\}$.
3. If $\gamma_s \notin \Gamma$, return $\{\text{REJECT} \times \text{"}\sqcup\text{"} \times R\}$.
4. Search the defined transitions (stored as $\{\{q_f \times \gamma_f\} \times \{q_t \times \gamma_t \times d\}\}$) to find one where $q_f = q_s \ \wedge \ \gamma_f = \gamma_s$.
5. If such a defined transition was found, return the transition $\{q_t \times \gamma_t \times d\}$.
6. If no such defined transition was found, return $\{\text{REJECT} \times \text{"}\sqcup\text{"} \times R\}$.

Given the constraint preventing multiple possible values for the same input pair of state and symbol, and the compiler's guarantee that if the developer does not specify a transition the compiler will provide one, our transition function will have exactly one possible result for each possible input pair of state and symbol. The developer can define some or all of these results, and the compiler will fill in the rest.

1 (d) A note about performance

My goal here is to demonstrate *how* to have a Turing Machine execute code. Performance, either time-wise or memory-wise, **is not a concern** for this specification. While the underlying implementation, in whatever form it takes, is

[4] My implementation prevents redeclaring transitions even if the new version is identical to the existing version. But, if an implementation allows providing multiple tuples that begin with the same $Q \times \Gamma$ values as long as the rest of the tuple is also identical, such an implementation would not prevent the transition function from being deterministic since there would still only be a single possible transition result.

likely to be restricted with memory constraints, Turing Machines are predicated on an infinitely long tape of symbols being available, and thus I assume that there is infinite memory available.[5] The set of supported states and the tape alphabet must also support an unbounded number of entries - while these are fixed for each individual configuration of a Turing Machine, they can be arbitrarily large.

Since the transition function of a Turing Machine only accepts two inputs, the current state name and the current symbol (or, in the case of a multi-tape Turing Machine, the current state name and the set of current symbols), to implement additional features extra information is stored in the name of the state, or in the symbol itself. The details of this will become clear in the coming pages, but generally, the requirement of support for an unbounded number of states and symbols is used as a substitute for "memory", allowing the machine to hold additional information.

That being said, in the implementation that is attached, I did run into performance issues in a few places, specifically dealing with the maximum call stack size, the maximum size of JavaScript Maps (jmrk 2019), and Chrome crashing as out of memory. These issues, when encountered, are merely places where the implementation does not properly match the requirements of this specification, not indications that the specification itself is flawed. A common cause of such issues was disabling some of the extra features that the compiler uses, forcing the expansion of the various shortcuts like defining transitions that target multiple symbols (or even serve as the default fallback transition) coupled with larger programs. My hope is that the descriptions here convince you that the shortcuts provide no new computational functionality, and thus that later, more complicated examples, only need to be tested with the initial set of shortcuts enabled, to avoid these performance pitfalls.

[5]Similarly, the underlying implementation is likely to be restricted by time constraints - no computer system will last forever, and even if a system did last until the heat death of the universe, the developer is unlikely to have the patience to wait that long.

2 Expanding functionality of single tape machines

Let us begin with a number of contrived program examples that, though not individually useful, serve to make concrete the extra features that are added to the TM. To begin with, description of programs will be given in three parts - a high-level explanation, a low-level description of states and transitions in prose, and pseudocode for those exact same states and transitions.

2 (a) Matching multiple symbols (and leaving symbols unchanged)

Our first three programs all do the same thing: given a string of input digits, replace the odd ones with a "1", leave the even ones unchanged, and then halt and accept upon reaching the end of the list. The three iterations serve to introduce the first two "extra features" that the compiler will ensure are available - the ability to define a transition for multiple symbols at once, and to define such a transition that doesn't change the symbols on the tape.

Program #1: Odd digit replacement (all manually)
Given a series of digits, for each odd digit replace it with a "1", for each even digit leave it unchanged, and upon encountering the first empty cell "␣", halt and accept.
In state q_{START}: • On symbol "1", replace with "1", move right, and change to state q_{START} • On symbol "3", replace with "1", move right, and change to state q_{START} • On symbol "5", replace with "1", move right, and change to state q_{START} • On symbol "7", replace with "1", move right, and change to state q_{START} • On symbol "9", replace with "1", move right, and change to state q_{START} • On symbol "2", replace with "2", move right, and change to state q_{START} • On symbol "4", replace with "4", move right, and change to state q_{START} • On symbol "6", replace with "6", move right, and change to state q_{START} • On symbol "8", replace with "8", move right, and change to state q_{START} • On symbol "0", replace with "0", move right, and change to state q_{START} • On symbol "␣", replace with "␣", move right, and change to state q_{ACCEPT}

```
TM.getState( "q_START" )
 .addTransition( "1" , "1" , R, "q_START" )
 .addTransition( "3" , "1" , R, "q_START" )
 .addTransition( "5" , "1" , R, "q_START" )
 .addTransition( "7" , "1" , R, "q_START" )
 .addTransition( "9" , "1" , R, "q_START" )
 .addTransition( "2" , "2" , R, "q_START" )
 .addTransition( "4" , "4" , R, "q_START" )
 .addTransition( "6" , "6" , R, "q_START" )
 .addTransition( "8" , "8" , R, "q_START" )
 .addTransition( "0" , "0" , R, "q_START" )
 .addTransition( EMPTY_CELL , EMPTY_CELL , R,
    STATE_ACCEPT )
```

Feature #1: Matching multiple symbols

It should be fairly clear right away that there is significant duplication of the transitions. Let's start with the odd digits: for each odd digit we define an identical transition, where the digit is replaced with a `"1"` and then the machine moves right and remains in the start state. Let's add a feature to avoid needing to duplicate these, by allowing the developer to define multiple transitions at the same time that all do the same thing (in this case, write a `"1"`, move the machine right, and remain in the start state).

Program #2: Odd digit replacement (match multiple)

Given a series of digits, for each odd digit replace it with a `"1"`, for each even digit leave it unchanged, and upon encountering the first empty cell `"␣"`, halt and accept.

In state q_{START}:

- On symbol `"2"`, replace with `"2"`, move right, and change to state q_{START}
- On symbol `"4"`, replace with `"4"`, move right, and change to state q_{START}
- On symbol `"6"`, replace with `"6"`, move right, and change to state q_{START}
- On symbol `"8"`, replace with `"8"`, move right, and change to state q_{START}
- On symbol `"0"`, replace with `"0"`, move right, and change to state q_{START}
- On symbol `"␣"`, replace with `"␣"`, move right, and change to state q_{ACCEPT}
- On any of the symbols ["1", "3", "5", "7", "9"], replace with `"1"`, move right, and change to state q_{START}

```
TM.getState( "q_START" )
 .addTransition(  "2"  ,  "2"  , R, "q_START" )
 .addTransition(  "4"  ,  "4"  , R, "q_START" )
 .addTransition(  "6"  ,  "6"  , R, "q_START" )
 .addTransition(  "8"  ,  "8"  , R, "q_START" )
 .addTransition(  "0"  ,  "0"  , R, "q_START" )
 .addTransition(   EMPTY_CELL    ,   EMPTY_CELL    ,   R,
    STATE_ACCEPT )
 .addTransition( ["1", "3", "5", "7", "9"],  "1"  , R, "q_START" )
```

At this point, when the developer defines new transitions, rather than simply providing tuples in the form

$$Q \times \Gamma \times Q \times \Gamma \times \{L, R\}$$

the developer can also specify that the single transition should apply to multiple target symbols, by providing an array of such symbols in Γ to target:

$$Q \times \Gamma[] \times Q \times \Gamma \times \{L, R\}$$

These two signatures can be combined with the following algebraic data type:

$$Q \times (\Gamma | \Gamma[]) \times Q \times \Gamma \times \{L, R\}$$

where $(\Gamma|\Gamma[])$ means that the second part of the tuple is *either* a single symbol in Γ or multiple such symbols.

While the developer now has a shortcut to *define* new transitions, the type of the actual transition function has not changed; during execution, the machine is only looking at a single symbol at a time (or in the case of multi-tape machines explained later on, a single symbol for each tape). In other words, the type of the transition function is still

$$Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\}$$

More precisely, referring back to our abstract function DefTransition, when passing in multiple target symbols to define multiple transitions at once, the behavior can be expressed as (where FromSymbols $\subset \Gamma$):

$$\begin{gathered}\textsc{DefTransition}(q_f, \textsc{FromSymbols}, q_t, \gamma_t, d) = \\ \forall \gamma_f \in \textsc{FromSymbols}, \textsc{DefTransition}(q_f, \gamma_f, q_t, \gamma_t, d)\end{gathered}$$

To be clear, the prohibitions on the definitions of multiple different transitions on the same target state and symbol apply to the underlying targets if multiple target symbols are provided, not the subset of targets itself. Two sets of transitions defined for a state using this multi-match feature, one for either of the symbols `[ "A", "B" ]` and the other for either of the symbols `[ "A", "C" ]`, conflict. Even though the actual subsets of Γ specified are different, there is a conflict because the transition for symbol `"A"` is defined twice.

However, there is significant duplication of the transitions. For each of the even digits, we define a transition that replaces the digit with itself, i.e., leaves the tape contents unchanged.[6] Let's add a feature to avoid needing to duplicate these too - the special constant `"NO_CHANGE"` will indicate in the pseudocode that the machine (or compiler) should treat the transition as one that doesn't change the contents of the tape.

Program #3: Odd digit replacement (match multiple, write or no change)
Given a series of digits, for each odd digit replace it with a `"1"`, for each even digit leave it unchanged, and upon encountering the first empty cell `"␣"`, halt and accept.
In state $q_{\texttt{START}}$: • On symbol `"␣"`, replace with `"␣"`, move right, and change to state $q_{\texttt{ACCEPT}}$ • On any of the symbols ["1", "3", "5", "7", "9"], replace with `"1"`, move right, and change to state $q_{\texttt{START}}$ • On any of the symbols ["2", "4", "6", "8", "0"], leave symbol unchanged, move right, and change to state $q_{\texttt{START}}$

`TM.getState( "`$q_{\texttt{START}}$`" )`
`.addTransition(` EMPTY_CELL , EMPTY_CELL , R, STATE_ACCEPT `)`
`.addTransition( ["1", "3", "5", "7", "9"], "1" , R, "`$q_{\texttt{START}}$`" )`
`.addTransition( ["2", "4", "6", "8", "0"],` NO_CHANGE, R, `"`$q_{\texttt{START}}$`" )`

At this point, when the developer defines new transitions, rather than specifying a specific symbol that the transition should write, the special constant `"NO_CHANGE"` can be used to indicate that the transition function should return a transition that writes the same symbol it matched against, i.e. the transition will not change the symbol on the tape. The developer defines transitions in the form:

$$Q \times (\Gamma | \Gamma[]) \times Q \times (\Gamma | \texttt{NO\_CHANGE}) \times \{L, R\}$$

Once again, while the developer has a new way to define transitions, the type of the actual transition function has not changed. When the transition function is called with some inputs $q \times \gamma$ (where $q \in Q$ and $\gamma \in \Gamma$), if the developer wants the result to be `"NO_CHANGE"`, the result from the transition function will just include the original γ that it was called with. Referring back to our abstract function DefTransition, this behavior can be expressed as (where FromSymbols $\subset \Gamma$):

[6] Technically we also do this for the `"1"` symbol but that can be handled with the other odd digits.

$$\textsc{DefTransition}(q_f, \textsc{FromSymbols}, q_t, \texttt{NO\_CHANGE}, d) = \\ \forall \gamma_f \in \textsc{FromSymbols}, \textsc{DefTransition}(q_f, \gamma_f, q_t, \gamma_f, d)$$

Note that this also allows the developer to use "NO_CHANGE" when defining a transition on a *single* symbol. It should be fairly clear that this provides no additional functionality that needs to be walked through - if a transition targets a single symbol and does not want to change anything on the tape, it is equivalent to that transition writing the single symbol it targeted. In other words:

$$\textsc{DefTransition}(q_f \times \gamma_f \times q_t \times \texttt{NO\_CHANGE} \times d) = \\ \textsc{DefTransition}(q_f \times \gamma_f \times q_t \times \gamma_f \times d)$$

2 (b) Fallback transitions

Next, let us consider another contrived set of programs with a new purpose: given an input of lowercase letters that may contain a period ("."), leave all letters before the first period unchanged, and replace all letters after the first period with periods. Upon reaching the end of the input, accept if there has been one or more periods, and otherwise reject.

Program #4: Letter replacement (match multiple)
Given a series of lowercase letters and periods, for each letter before the first period leave it unchanged, and for each letter after the first period replace it with a period. Upon encountering the first empty cell ("␣") halt, accepting if any period were found and rejecting otherwise.
In state $q_{\texttt{START}}$: • On symbol ".", replace with ".", move right, and change to state $q_{\texttt{replacing}}$ • On symbol "␣", replace with "␣", move right, and change to state $q_{\texttt{REJECT}}$ • On any of the symbols ["a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m", "n", "o", "p", "q", "r", "s", "t", "u", "v", "w", "x", "y", "z"], leave symbol unchanged, move right, and change to state $q_{\texttt{START}}$ In state $q_{\texttt{replacing}}$: • On symbol "␣", replace with "␣", move right, and change to state $q_{\texttt{ACCEPT}}$ • On any of the symbols ["a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m", "n", "o", "p", "q", "r", "s", "t", "u", "v", "w", "x", "y", "z", "."], replace with ".", move right, and change to state $q_{\texttt{replacing}}$

```
TM.getState( "q_START" )
 .addTransition( "." , "." , R, "q_replacing" )
 .addTransition( EMPTY_CELL , EMPTY_CELL , R,
    STATE_REJECT )
 .addTransition( ["a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l",
    "m", "n", "o", "p", "q", "r", "s", "t", "u", "v", "w", "x", "y", "z"],
    NO_CHANGE, R, "q_START" )

TM.getState( "q_replacing" )
 .addTransition( EMPTY_CELL , EMPTY_CELL , R,
    STATE_ACCEPT )
 .addTransition( ["a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l",
    "m", "n", "o", "p", "q", "r", "s", "t", "u", "v", "w", "x", "y", "z", "."],
    "." , R, "q_replacing" )
```

Feature #3: Wildcard matching

Though using multi-match transitions has avoided the developer needing to define the 53 transitions (two for each lowercase letter and one for the "." in the replacing state) separately, it is still tedious to manually list out all of the lowercase letters. It would be even worse if the tape alphabet included uppercase letters, digits, or other options that should be treated the same. In this case, since the tape alphabet is limited to the period, the empty symbol, and the letters, in the start state we know that after defining the handling for both the period and empty symbol, *for any other symbol in the tape alphabet* we want to leave it unchanged. Similarly, for the replacing state, after defining the handling for the empty symbol we know that *for any other symbol in the tape alphabet* we want to replace it with a period. Let's add a feature to avoid needing to list all these symbols, that tells the compiler (or machine) that a transition should be used for any symbol that does not already have a specific transition defined. The special constant "`WILDCARD`" will indicate this in the pseudocode.

Like with matching multiple symbols, it would also be useful for the wildcard or fallback transitions to be able to specify that no change should be made. We will reuse the same "`NO_CHANGE`" constant in the pseudocode for this too. Unlike for matching multiple symbols, the functionality to match for a transition that writes a symbol and a transition that makes no change to the tape are not separated into two different features that get introduced separately - it should be pretty clear that the no-change version of a transition is no more powerful than a transition that does write a symbol to the tape. If the wildcard feature is not supported, the compiler will make use of a transition that matches multiple symbols, which we know can handle no-change transitions just fine.

Program #5: Letter replacement (wildcard)

Given a series of lowercase letters and periods, for each letter before the first period leave it unchanged, and for each letter after the first period replace it with a period. Upon encountering the first empty cell ("␣") halt, accepting if any period were found and rejecting otherwise.

In state $q_{\texttt{START}}$:

- On symbol ".", replace with ".", move right, and change to state $q_{\texttt{replacing}}$
- On symbol "␣", replace with "␣", move right, and change to state $q_{\texttt{REJECT}}$
- On any other symbol, leave symbol unchanged, move right, and change to state $q_{\texttt{START}}$

In state $q_{\texttt{replacing}}$:

- On symbol "␣", replace with "␣", move right, and change to state $q_{\texttt{ACCEPT}}$
- On any other symbol, replace with ".", move right, and change to state $q_{\texttt{replacing}}$

```
TM.getState( "q_START" )
 .addTransition( "." , "." , R, "q_replacing" )
 .addTransition( EMPTY_CELL , EMPTY_CELL , R,
    STATE_REJECT )
 .addTransition( WILDCARD, NO_CHANGE, R, "q_START" )

TM.getState( "q_replacing" )
 .addTransition( EMPTY_CELL , EMPTY_CELL , R,
    STATE_ACCEPT )
 .addTransition( WILDCARD, "." , R, "q_replacing" )
```

At this point, when the developer defines new transitions, a transition can be created that targets any symbol in Γ, creating a fallback transition when no others are defined for the given combination of state and symbol. The developer defines transitions in the form:

$$Q \times (\Gamma | \Gamma[] | \texttt{WILDCARD}) \times Q \times (\Gamma | \texttt{NO\_CHANGE}) \times \{L, R\}$$

such that

$$\begin{gathered}\textsc{DefTransition}(q_f, \texttt{WILDCARD}, q_t, g_t, d) = \\ \textsc{FromSymbols} = \{\gamma \| \gamma \in \Gamma, \neg(\textsc{IsTransitionDefined}(q_f, \gamma))\} \\ \textsc{DefTransition}(q_f, \textsc{FromSymbols}, q_t, g_t, d)\end{gathered}$$

where g_t is either a specific symbol γ_t or the constant `NO_CHANGE`. The function IsTransitionDefined simply checks if a transition is already defined. Note that once a wildcard transition has been defined, all symbols for the state have had transitions defined, and no further transitions can be added. Thus, a wildcard transition must always be defined last among the transitions for a state.

2 (c) No movement

Next, let us consider another simple contrived program: given an input of lowercase letters that may contain a "0", halt and accept if the input contains at least one "0" **with the head of the machine pointing to the first "0"**, otherwise reject.[7] Since we at the moment assume that every transition *has* to move the machine head left or right one cell, this could look like:

Program #6: Find first "0" (manual extra state)

Given a series of lowercase letters and "0"s, halt and accept with the head of the machine pointing to the first "0", or reject if there are none.

In state $q_{\texttt{START}}$:

- On symbol "0", replace with "0", move right, and change to state $q_{\texttt{goLeftAndAccept}}$
- On symbol "␣", replace with "␣", move right, and change to state $q_{\texttt{REJECT}}$
- On any other symbol, leave symbol unchanged, move right, and change to state $q_{\texttt{START}}$

In state $q_{\texttt{goLeftAndAccept}}$:

- On any symbol, leave symbol unchanged, move left, and change to state $q_{\texttt{ACCEPT}}$

```
TM.getState( "q_START" )
 .addTransition( "0" , "0" , R, "q_goLeftAndAccept" )
 .addTransition( EMPTY_CELL , EMPTY_CELL , R,
    STATE_REJECT )
 .addTransition( WILDCARD, NO_CHANGE, R, "q_START" )

TM.getState( "q_goLeftAndAccept" )
 .addTransition( WILDCARD, NO_CHANGE, L, STATE_ACCEPT )
```

Feature #4: No movement

Though it is fairly trivial to define the single helper state $q_{\texttt{goLeftAndAccept}}$ in this case, once we want to add moving multiple cells at a time (down below) it will get much more complicated. So let's use transitions with no movement as an easy case. Here, we want to allow the developer to specify that, in the start state when a "0" is found, instead of moving right and then left, the machine

[7] The head of the machine is "pointing" at a symbol when the machine is currently examining that symbol on the tape. This is also referred to as the machine, or the heading of the machine, being "over" the symbol, "at" the symbol, or "looking at" the symbol, or other such terms - there is no difference in the meaning of the various terms.

should immediately accept without moving the tape head. If the machine is unable to do so, the compiler will provide the needed functionality.

This is the first extra feature that may require the compiler to introduce additional state names (in this case it would have been $q_{\texttt{goLeftAndAccept}}$) and we need to ensure that the compiler-produced states don't conflict with any states that the developer tries to define themselves. Since we are dealing purely with the world of "symbols" rather than being restricted to the Latin alphabet or ASCII characters, or even the larger (but still finite) set of Unicode characters, we can simply have the compiler *invent* some symbol that isn't used in any of the states that the developer defines, and make use of that.[8] Only states created by the compiler would contain that symbol in their names, and *all* compiler-created states must have that symbol somewhere in their names, to prevent conflicts with user-provided state names.

To avoid confusion, here I will always represent that symbol using the lowercase Greek omega ("ω"), but this is simply for documentation purposes - if the developer wanted to write a program that used states that contained an "ω" in their names, the compiler would just pick some other symbol.[9]

Thus, we can avoid conflicts between state names generated by the compiler and those provided by the developer by surrounding any developer-provided inputs with a "ω" (or the other invented symbol). Internally, the compiler will ensure that it prevents conflicts between state names that it generates.

If the developer is given access to a no-move feature, but the underlying machine lacks such a feature, the compiler would implement the functionality. We want the compiler to produce a program almost identical to program #6 from page 14, but with a slightly different name for the helper state that just always goes left:

[8] There are an infinite number of possible symbols available for use.

[9] Since my implementation is written in JavaScript and stores the state names as strings, I cannot simply *invent* a new symbol to use. Instead, I simply use a string of literal "ω"s. I start with one, and if the developer wants to add a state that includes "ω", the indicator for internal states is changed to "ωω". If the developer wants to add a state that includes two of the symbols, the internal marker uses three, and so on. Thus, I can ensure that only states added by the compiler contain however many "ω"s indicate internal states. Note that each time the internal marker changes, all existing uses are updated, so the order in which the states were defined does not prevent us from avoiding state name conflicts.

Program #7: Find first "0" (*compiler view* - extra state)

Given a series of lowercase letters and "0"s, halt and accept with the head of the machine pointing to the first "0", or reject if there are none.

In state $q_{\texttt{START}}$:

- On symbol "0", replace with "0", move right, and change to state $q_{\texttt{ω\_1L\_ω\_qAccept\_ω}}$
- On symbol "␣", replace with "␣", move right, and change to state $q_{\texttt{REJECT}}$
- On any other symbol, leave symbol unchanged, move right, and change to state $q_{\texttt{START}}$

In state $q_{\texttt{ω\_1L\_ω\_qAccept\_ω}}$:

- On any symbol, leave symbol unchanged, move left, and change to state $q_{\texttt{ACCEPT}}$

```
TM.getState( "q_START" )
 .addTransition( "0" , "0" , R, "q_ω_1L_ω_qAccept_ω" )
 .addTransition( EMPTY_CELL , EMPTY_CELL , R,
    STATE_REJECT )
 .addTransition( WILDCARD, NO_CHANGE, R, "q_START" )

TM.getState( "q_ω_1L_ω_qAccept_ω" )
 .addTransition( WILDCARD, NO_CHANGE, L, STATE_ACCEPT )
```

We want the compiler to generate helper states where the name of each form is in the format `ω_{movement remaining}_ω_{target state}_ω`, which serves to make them consistent when thinking ahead a bit regarding moving multiple cells at once. If the compiler has the ability to create these states and transitions such that the developer can define transitions with no movement (and we now declare that yes, the compiler should have that ability), then the program that the developer would right would be as simple as:

Program #8: Find first "0" (using no-move feature)

Given a series of lowercase letters and "0"s, halt and accept with the head of the machine pointing to the first "0", or reject if there are none.

In state $q_{\texttt{START}}$:

- On symbol "0", replace with "0", do not move the tape head, and change to state $q_{\texttt{ACCEPT}}$
- On symbol "␣", replace with "␣", move right, and change to state $q_{\texttt{REJECT}}$
- On any other symbol, leave symbol unchanged, move right, and change to state $q_{\texttt{START}}$

```
TM.getState( "q_START" )
 .addTransition( "0" , "0" , N, STATE_ACCEPT )
 .addTransition( EMPTY_CELL , EMPTY_CELL , R,
    STATE_REJECT )
 .addTransition( WILDCARD, NO_CHANGE, R, "q_START" )
```

At this point, when the developer defines new transitions, a transition can be created that results in no movement on the tape. The developer defines transitions in the form:

$$Q \times (\Gamma|\Gamma[]|\texttt{WILDCARD}) \times Q \times (\Gamma|\texttt{NO\_CHANGE}) \times \{L, R, N\}$$

where the "N" option indicates that the head of the tape should not move in either direction.

The addition of this new feature is also the first time that the signature of the underlying transition function changes. Rather than simply being in the form

$$Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\}$$

transitions can now be returned that specify that no movement should be performed:

$$Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R, N\}$$

If the underlying machine does not support such a feature, the compiler will add extra states and transitions to simulate this feature. However, the developer does not need to know about whether the underlying machine actually supports the feature - they simply make use of no-move transitions, and assume that the machine has such a feature. If the machine does not actually support the feature, the compiler transparently provides the functionality.[10]

It is the responsibility of the compiler to prevent the states that it generates for each feature from conflicting with each other or with the states generated by other features. To aid in confirming that the names do not conflict, I include an appendix listing the different forms of state names that the compiler will generate for various features. Since all of the state names will include a "ω" (or other invented symbol) to prevent conflicts between compiler-generated state names and developer-provided state names, we only need to prevent conflicts between different compiler-generated state names.

[10]Unfortunately, in English "transparent" can refer to either something that can be seen through, such as a "transparent pane of glass", or something that is easy to detect, such as a "transparent attempt". In this case, I use "transparent" with the former meaning - the compiler may silently insert extra logic to implement a feature, but the developer essentially "sees through" that extra logic and just looks at what it accomplishes (in this case a transition that does not move the head of the tape).

Looking at our abstract function DefTransition, if q_h is the helper state that gets created, then

$$\begin{gathered}\textsc{DefTransition}(q_f, g_f, q_t, g_t, N) = \\ \textsc{DefTransition}(q_f, g_f, q_h, g_t, R) \\ \textsc{DefTransition}(q_h, \texttt{WILDCARD}, q_t, \texttt{NO\_CHANGE}, L)\end{gathered}$$

Where g_f is either a single symbol γ_f, an array of multiple symbols, or itself `WILDCARD`, and g_t is either a single symbol γ_t or the constant `NO_CHANGE`.

To be clear, while moving left and then right, or moving right and then left, might seem to both work as ways to implement the no-move feature, moving left and then right is not actually an option. We *need* the no-movement transitions to move right and then left. Consider the following program that the compiler would have generated if we moved left and then right:

Program #9: Find first "0" (*compiler view* - broken)

Given a series of lowercase letters and "0"s, halt and accept with the head of the machine pointing to the first "0", or reject if there are none.

In state $q_{\texttt{START}}$:

- On symbol "0", replace with "0", move left, and change to state $q_{\texttt{ω\_1R\_ω\_accept\_ω}}$
- On symbol "␣", replace with "␣", move right, and change to state $q_{\texttt{REJECT}}$
- On any other symbol, leave symbol unchanged, move right, and change to state $q_{\texttt{START}}$

In state $q_{\texttt{ω\_1R\_ω\_accept\_ω}}$:

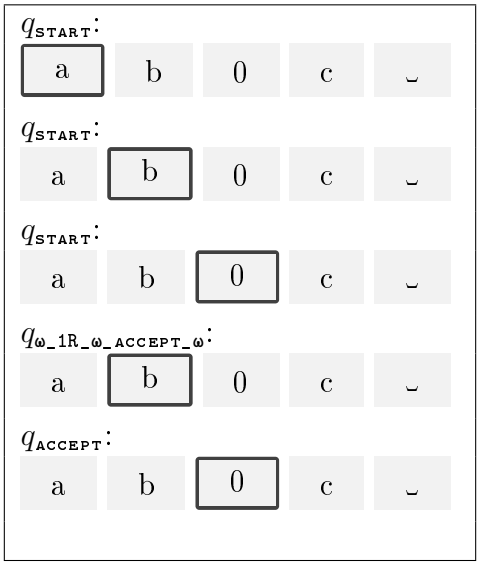
- On any symbol, leave symbol unchanged, move right, and change to state $q_{\texttt{ACCEPT}}$

```
TM.getState( "q_START" )
 .addTransition( "0" , "0" , L, "q_ω_1R_ω_accept_ω" )
 .addTransition( EMPTY_CELL , EMPTY_CELL , R,
    STATE_REJECT )
 .addTransition( WILDCARD, NO_CHANGE, R, "q_START" )

TM.getState( "q_ω_1R_ω_accept_ω" )
 .addTransition( WILDCARD, NO_CHANGE, R, STATE_ACCEPT )
```

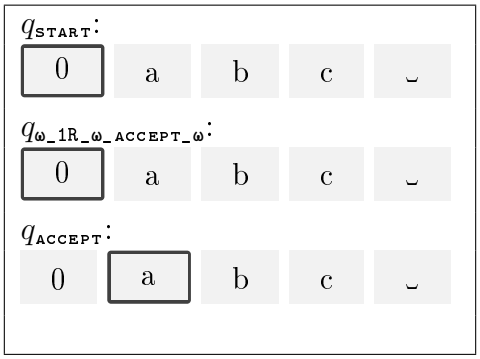
On most inputs this program might work fine. For example, if the input was `"ab0c"`, the execution would go as follows:[11]

Program #9 with input `"ab0c"`:



In this case, the machine would work. But, what if the input *started* with the first `"0"`, like in the input `"0abc"`? The execution would go as follows:

Program #9 with input `"0abc"`:



In the second case, the result is clearly incorrect: the machine did not end with the head of the tape on the `"0"`. The failure is caused by the fact that when the machine tries to move left while looking at the first cell, the tape head doesn't actually move (Sipser 2013, 168). Thus, a left-then-right sometimes results in only moving to the right. Instead of needing to worry about trying to add special handling for detecting the start of the tape, the

[11] Each row displays a step of the execution. The state the machine is in *before* processing the new symbol is listed before the tape contents, and the tape contents are shown in individual boxes to represent the different cells. The cell where the head of the machine is current located is surrounded with a bold box to indicate that it is the symbol currently being pointed to. An infinite number of empty cells always exist to the right of the cells shown, but they are not all shown.

compiler instead should just always implement a no-move transition as a move right and then a move left.

The feature to not move the head of the tape also serves to introduce what will be a key technique for further features: storing additional data in state names. Regardless of how the developer is allowed to defined new transitions, the compiler must be able to support a machine that has none of the extra features, i.e. it must support a transition function in the form

$$Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\}$$

To be able to expand the functionality that the developer can use to define new transitions, even when the underlying machine lacks that functionality, we need to add some custom symbols to Γ, some custom states to Q, or both. Here, the compiler generates state names indicating what transition is still needed before finally ending in the state the developer wanted to be in. For other features, the compiler can generate other state names. As long as these names are determined at compile time, the compiler is free to store as much information as it wants in the names.

2 (d) Moving multiple cells

Next, let us consider an example where we want to move over multiple cells at once: given an input of lowercase letters that may include commas, try to find a comma that is followed three cells later by another comma. If one of these pairs of commas is found, halt and accept with the head of the machine on the first of the two commas of the first pair. If no such pair of commas is found, reject.

This can be implemented with the following high-level algorithm:

- On a comma, go right three cells.
  - If the cell three spaces over contains a comma, go left three cells to the original comma and accept.
  - Otherwise go left two cells (i.e. to the cell immediately after the first comma) and resume processing.
- If the cell is empty, the entire input has been processed and no pairs were found, so reject.
- On any other symbol, simply move to the right to check the next symbol.

Translated into specific states and instructions for a program, this would look like the following, with the ideal compiler-provided state names used for the transitions that move multiple cells:

Program #10: Find start of comma pair (*compiler view* - extra states)

Given a series of lowercase letters and commas, halt and accept with the head of the machine pointing to the first comma that is followed three cells later by another comma, or reject if there are no such pairs.

In state $q_{\texttt{START}}$:
- On symbol ",", replace with ",", move right, and change to state $q_{\texttt{ω\_2R\_ω\_qcheckComma\_ω}}$
- On symbol "␣", replace with "␣", move right, and change to state $q_{\texttt{REJECT}}$
- On any other symbol, leave symbol unchanged, move right, and change to state $q_{\texttt{START}}$

In state $q_{\texttt{ω\_2R\_ω\_qcheckComma\_ω}}$:
- On any symbol, leave symbol unchanged, move right, and change to state $q_{\texttt{ω\_1R\_ω\_qcheckComma\_ω}}$

In state $q_{\texttt{ω\_1R\_ω\_qcheckComma\_ω}}$:
- On any symbol, leave symbol unchanged, move right, and change to state $q_{\texttt{checkComma}}$

In state $q_{\texttt{checkComma}}$:
- On symbol ",", replace with ",", move left, and change to state $q_{\texttt{ω\_2L\_ω\_qAccept\_ω}}$
- On any other symbol, leave symbol unchanged, move left, and change to state $q_{\texttt{ω\_1L\_ω\_qstart\_ω}}$

In state $q_{\texttt{ω\_2L\_ω\_qAccept\_ω}}$:
- On any symbol, leave symbol unchanged, move left, and change to state $q_{\texttt{ω\_1L\_ω\_qAccept\_ω}}$

In state $q_{\texttt{ω\_1L\_ω\_qAccept\_ω}}$:
- On any symbol, leave symbol unchanged, move left, and change to state $q_{\texttt{ACCEPT}}$

In state $q_{\texttt{ω\_1L\_ω\_qstart\_ω}}$:
- On any symbol, leave symbol unchanged, move left, and change to state $q_{\texttt{START}}$

```
TM.getState( "qSTART" )
 .addTransition( "," , "," , R, "qω_2R_ω_qcheckComma_ω" )
 .addTransition( EMPTY_CELL , EMPTY_CELL , R,
    STATE_REJECT )
 .addTransition( WILDCARD, NO_CHANGE, R, "qSTART" )
```

```
TM.getState( "q_ω_2R_ω_qcheckComma_ω" )
 .addTransition( WILDCARD, NO_CHANGE, R, "q_ω_1R_ω_qcheckComma_ω" )

TM.getState( "q_ω_1R_ω_qcheckComma_ω" )
 .addTransition( WILDCARD, NO_CHANGE, R, "q_checkComma" )

TM.getState( "q_checkComma" )
 .addTransition( "," , "," , L, "q_ω_2L_ω_qAccept_ω" )
 .addTransition( WILDCARD, NO_CHANGE, L, "q_ω_1L_ω_qstart_ω" )

TM.getState( "q_ω_2L_ω_qAccept_ω" )
 .addTransition( WILDCARD, NO_CHANGE, L, "q_ω_1L_ω_qAccept_ω" )

TM.getState( "q_ω_1L_ω_qAccept_ω" )
 .addTransition( WILDCARD, NO_CHANGE, L, STATE_ACCEPT )

TM.getState( "q_ω_1L_ω_qstart_ω" )
 .addTransition( WILDCARD, NO_CHANGE, L, "q_START" )
```

Feature #5: Multiple cell movement

To move three cells to the right, we move one cell over, and transition to a helper state that will move over another two cells. To move two cells over, we move one cell over, and then use another helper state to move the last time. The same works moving to the left. These states and transitions can be trivially generated by the compiler, as they follow a clear formula: to move v cells, move once and switch to a state that will handle the remaining $v - 1$ movements. Thus, the developer should be able to write their program as:

Program #11: Find start of comma pair (using multi-move feature)
Given a series of lowercase letters and commas, halt and accept with the head of the machine pointing to the first comma that is followed three cells later by another comma, or reject if there are no such pairs.
In state q_{START}: • On symbol ",", replace with ",", move right three cells, and change to state $q_{\text{checkComma}}$ • On symbol "␣", replace with "␣", move right, and change to state q_{REJECT} • On any other symbol, leave symbol unchanged, move right, and change to state q_{START}

In state $q_{\texttt{checkComma}}$:

- On symbol ",", replace with ",", move left three cells, and change to state $q_{\textsc{accept}}$
- On any other symbol, leave symbol unchanged, move left two cells, and change to state $q_{\textsc{start}}$

```
TM.getState( "q_START" )
 .addTransition( "," , "," , R(3), "q_checkComma" )
 .addTransition( EMPTY_CELL , EMPTY_CELL , R,
    STATE_REJECT )
 .addTransition( WILDCARD, NO_CHANGE, R, "q_START" )

TM.getState( "q_checkComma" )
 .addTransition( "," , "," , L(3), STATE_ACCEPT )
 .addTransition( WILDCARD, NO_CHANGE, L(2), "q_START" )
```

Like with no-move transitions, with transitions that move over multiple cells, in addition to changing the signature of how the developer defines new transitions, the signature of the actual transition function needs to be updated. The developer defines transitions in the form:

$$Q \times (\Gamma|\Gamma[]|\texttt{WILDCARD}) \times Q \times (\Gamma|\texttt{NO\_CHANGE}) \times \{L, R, N, L(v), R(v)\}$$

meaning that a transition can include moving any specified number of cells to the left or right.[12] The actual transition function is also now:

$$Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R, N, L(v), R(v)\}$$

If the underlying machine does not support moving the head of the tape over multiple cells at once, the compiler will add extra states and transitions to simulate this feature, just like the compiler did for the no-move transitions. Again, the developer does not need to know about whether the underlying machine actually supports the feature - they simply make use of multi-move transitions, and if needed the compiler transparently provides the functionality.

Looking at our abstract function DefTransition, if v is the magnitude of the movement in the transition and q_h is the *first* helper state to be created, then

$$v > 1, \textsc{DefTransition}(q_f, g_f, q_t, g_t, d_v) =$$
$$\textsc{DefTransition}(q_f, g_f, q_h, g_t, d_1)$$
$$\textsc{DefTransition}(q_h, \texttt{WILDCARD}, q_t, \texttt{NO\_CHANGE}, d_{v-1})$$

[12] To be absolutely clear: the number of cells specified **must be** a whole number greater than zero, and *should* be greater than one. If the number specified is exactly one, the feature would still work but provides no advantage over the normal way of indicating moving the head of the tape one cell to the left or to the right.

Where g_f is either a single symbol γ_f, an array of multiple symbols, or itself `WILDCARD`, g_t is either a single symbol γ_t or the constant `NO_CHANGE`, d_1 is movement in the same direction as d_v by a single cell, and d_{v-1} is movement in the same direction as d_v but by one fewer cell than v. The implementation of this feature is recursive - after adding the first helper state, a new transition is defined with d_{v-1}, which (if $v \neq 2$) will result in another helper state to implement that transition, and so on until eventually the recursive call d_{v-1} is movement of a single cell left or right.

Now that the compiler supports moving over multiple cells at once, it would be a good time to mention that, when I write "move the tape head left" or "move the tape head right" without specifying a number of cells to move, the default is to move only a single cell in the specified direction. Without the multi-move feature, this would not have caused confusion, since moving a single cell was the only option, but now that it is possible to move over multiple cells at once it bears mentioning.

2 (e) Shifting cells over

For our next contrived example, let us consider a case where there is a series of `"0"`s and `"1"`s given as the input. The goal is to insert an extra `"1"` right before the first symbol in the input, for example an input of `"010"` would be transformed into `"1010"`. An empty input should just have the `"1"`. Ideally, we want to be able to just say "shift the current value of the cell to the right" when pointing at last cell value, and then the one before it, and so on, eventually (after shifting over the first symbol) writing a `"1"` at the start. But what might that look like if the machine didn't support such a feature?

The high-level algorithm for this would be

- If the first cell is empty, write a `"1"` and accept.
- Otherwise, go right to the first empty cell.
- For each cell before the first empty one, beginning from the end, move it to the right by erasing the current value and writing it in the cell directly to the right. Then go left two cells - once to reach the cell just erased, and once to reach the cell before that, which will be the next cell that should be shifted over.
- After moving the last cell value (i.e. when moving left two cells led to an empty cell) write a `"1"` in the now-blank leftmost cell, and accept.

Rather than first demonstrating this implementation with developer-provided states that are used to implement the shift, and then have another program that just uses the compiler-provided names but has the same logic, let us just start by looking at what the compiler shoud produce to implement this feature if it is not available in the underlying machine:

Program #12: Insert "1" before input (*compiler view* - extra states)

Given a series of "0"s and "1"s, insert an extra "1" before the input, shifting all symbols to the right.

In state $q_{\texttt{START}}$:
- On symbol "␣", replace with "1", do not move the tape head, and change to state $q_{\texttt{ACCEPT}}$
- On any other symbol, leave symbol unchanged, move right, and change to state $q_{\texttt{findEnd}}$

In state $q_{\texttt{findEnd}}$:
- On symbol "␣", replace with "␣", move left, and change to state $q_{\texttt{doShifts}}$
- On any other symbol, leave symbol unchanged, move right, and change to state $q_{\texttt{findEnd}}$

In state $q_{\texttt{doShifts}}$:
- On symbol "␣", replace with "1", do not move the tape head, and change to state $q_{\texttt{ACCEPT}}$
- On symbol "0", replace with "␣", move right, and change to state $q_{\texttt{ω\_paste0\_ω\_2L\_ω\_qdoShifts\_ω}}$
- On symbol "1", replace with "␣", move right, and change to state $q_{\texttt{ω\_paste1\_ω\_2L\_ω\_qdoShifts\_ω}}$

In state $q_{\texttt{ω\_paste0\_ω\_2L\_ω\_qdoShifts\_ω}}$:
- On any other symbol, replace with "0", move left two cells, and change to state $q_{\texttt{doShifts}}$

In state $q_{\texttt{ω\_paste1\_ω\_2L\_ω\_qdoShifts\_ω}}$:
- On any other symbol, replace with "1", move left two cells, and change to state $q_{\texttt{doShifts}}$

```
TM.getState( "qSTART" )
 .addTransition( EMPTY_CELL ,  "1"  , N, STATE_ACCEPT )
 .addTransition( WILDCARD, NO_CHANGE, R,  "qfindEnd" )

TM.getState( "qfindEnd" )
 .addTransition( EMPTY_CELL , EMPTY_CELL , L,  "qdoShifts" )
 .addTransition( WILDCARD, NO_CHANGE, R,  "qfindEnd" )

TM.getState( "qdoShifts" )
 .addTransition( EMPTY_CELL ,  "1"  , N, STATE_ACCEPT )
 .addTransition(  "0"  , EMPTY_CELL , R,  "qω_paste0_ω_2L_ω_qdoShifts_ω" )
 .addTransition(  "1"  , EMPTY_CELL , R,  "qω_paste1_ω_2L_ω_qdoShifts_ω" )
```

```
TM.getState( "q_ω_paste0_ω_2L_ω_qdoShifts_ω" )
 .addTransition( WILDCARD,  "0" , L(2),  "q_doShifts" )

TM.getState( "q_ω_paste1_ω_2L_ω_qdoShifts_ω" )
 .addTransition( WILDCARD,  "1" , L(2),  "q_doShifts" )
```

This can be tricky to visualize, so let's step through a concrete example with the input "010":

Program #12 with input "010":

q_{START}:

0	1	0	␣	␣

q_{findEnd}:

0	**1**	0	␣	␣

q_{findEnd}:

0	1	**0**	␣	␣

q_{findEnd}:

0	1	0	**␣**	␣

q_{doShifts}:

0	1	**0**	␣	␣

$q_{\text{ω\_paste0\_ω\_2L\_ω\_qdoShifts\_ω}}$:

0	1	␣	**␣**	␣

q_{doShifts}:

0	**1**	␣	0	␣

$q_{\text{ω\_paste1\_ω\_2L\_ω\_qdoShifts\_ω}}$:

0	␣	**␣**	0	␣

q_{doShifts}:

0	␣	1	0	␣

$q_{\text{ω\_paste0\_ω\_2L\_ω\_qdoShifts\_ω}}$:

␣	**␣**	1	0	␣

q_{doShifts}:

␣	0	1	0	␣

q_{ACCEPT}:

1	0	1	0	␣

This feature can take advantage of the fact that, when at the leftmost cell, moving left leaves the tape head looking at the same cell as before (Sipser 2013, 168). In this case, moving left by two cells when looking at the second cell accordingly results in the head ending up on the first cell. Since the input does not contain any empty cells at the start, finding an empty cell after moving left twice must indicate the start of the input, where the developer wants to write the "1".

Feature #6: Cell-shifting gadget

As can be seen from the manual set of states shown above, each cell value that needs to be shifted requires a dedicated state for it reflecting what symbol to write ("paste"), what subsequent movement to perform, and what final state to end up in:

- "ω_paste0_ω_2L_ω_qdoShifts_ω"
- "ω_paste1_ω_2L_ω_qdoShifts_ω"

While shifting an input containing just "0"s and "1"s isn't too hard, imagine writing the states and transitions manually for shifting all 26 lowercase Latin letters! It would quickly become tedious, so let's have the compiler do the work for us.

This is the first feature that I label a "gadget" - it is not merely a new way to declare the symbols that a transition targets (such as multiple symbols or a wildcard fallback), nor what symbol the transition should write (no change to the existing symbol), nor the movement of the machine afterward (stay in place or move multiple cells). It creates extra states which modify the contents of the tape, and accordingly deserves a different label, hence the term "gadget". More generally, I use "gadget" to indicate a feature that affects more than one cell, but this is merely a labeling practice - every "gadget" is really just a feature that I don't expect any real machine to have support for.

Let the gadget be used as follows:

- The gadget shifts the value of the starting cell exactly one cell to the right.
- After running, the starting cell is left with an "␣".
- The gadget is told what movement to make after the shift, relative to the cell to the right of the starting cell (i.e. relative to the cell into which the value was shifted).
- The gadget is told what state to end in after completion.

It is expected that the cell to the right of the starting cell (i.e. the tape cell where the value of the starting cell will be written) will be empty or otherwise known to the developer; regardless of that cell's initial value it will be overwritten with the value from the starting cell.

With this gadget, the developer could write the same program above as:

Program #13: Insert "1" before input (cell-shifting gadget; "0"s and "1"s)

Given a series of "0"s and "1"s, insert an extra "1" before the input, shifting all symbols to the right.

In state $q_{\texttt{START}}$:

- On symbol "␣", replace with "1", do not move the tape head, and change to state $q_{\texttt{ACCEPT}}$
- On any other symbol, leave symbol unchanged, move right, and change to state $q_{\texttt{findEnd}}$

In state $q_{\texttt{findEnd}}$:

- On symbol "␣", replace with "␣", move left, and change to state $q_{\texttt{doShifts}}$
- On any other symbol, leave symbol unchanged, move right, and change to state $q_{\texttt{findEnd}}$

In state $q_{\texttt{doShifts}}$:

- On symbol "␣", replace with "1", do not move the tape head, and change to state $q_{\texttt{ACCEPT}}$
- On symbol "0", replace with "␣", move right, replace the symbol there with "0", and then move left two cells, ending in state $q_{\texttt{doShifts}}$
- On symbol "1", replace with "␣", move right, replace the symbol there with "1", and then move left two cells, ending in state $q_{\texttt{doShifts}}$

```
TM.getState( "q_START" )
 .addTransition( EMPTY_CELL ,  "1"  , N, STATE_ACCEPT )
 .addTransition( WILDCARD, NO_CHANGE, R, "q_findEnd" )

TM.getState( "q_findEnd" )
 .addTransition( EMPTY_CELL , EMPTY_CELL , L, "q_doShifts" )
 .addTransition( WILDCARD, NO_CHANGE, R, "q_findEnd" )

TM.getState( "q_doShifts" )
 .addTransition( EMPTY_CELL ,  "1"  , N, STATE_ACCEPT )
 .addTransitionGadget(   "0"  , "SHIFT", L(2), "q_doShifts" )
 .addTransitionGadget(   "1"  , "SHIFT", L(2), "q_doShifts" )
```

To add gadgets, the pseudocode shown above makes use of a new method, "addTransitionGadget()", which is used to add a transition that is based on a gadget feature. The second parameter is the name of the gadget, in this case "SHIFT". The type of the tuples that the developer uses to define transitions also changes; the new type is given and explained at the end of this subsection.

To implement this gadget, the compiler generates state where the names of the states are in the format ω_paste{symbol}_ω_{movement}_ω_{target

`state}_ω`; no other compiler feature should generate such a state name. Just as the developer-provided state names are surrounded with "`ω`" to prevent conflicts, so too must the developer-provided symbols be sanitized to prevent conflicts.

To support "pasting" in an empty cell, and later for other places where a symbol is included in the state name, the compiler should support representing the empty cell value in a state name without conflicting with any other symbol in the tape alphabet. In the examples below, it is represented as "`{e}`", but as always, if the tape alphabet contains a symbol that looks like that, the compiler can choose any other representation it wants.[13]

This certainly reduced the duplication of the developer needing to write a dedicated state for each value that could be shifted. However, there is still some duplication of needing to write each usage of the gadget. Consider, for example, if we wanted to handle an input alphabet of all ten digits:

Program #14: Insert "1" before input (cell-shifting gadget; all digits)
Given a series of digits, insert an extra "1" before the input, shifting all symbols to the right.
In state $q_{\texttt{START}}$: • On symbol "␣", replace with "1", do not move the tape head, and change to state $q_{\texttt{ACCEPT}}$ • On any other symbol, leave symbol unchanged, move right, and change to state $q_{\texttt{findEnd}}$ In state $q_{\texttt{findEnd}}$: • On symbol "␣", replace with "␣", move left, and change to state $q_{\texttt{doShifts}}$ • On any other symbol, leave symbol unchanged, move right, and change to state $q_{\texttt{findEnd}}$

[13] In my implementation, instead of trying to change how the empty symbol is represented, if the developer's symbol is "`{e}`" it gets converted (in state names and other displays) to "`S:{e}`". To avoid conflicts on that symbol, the transformation is more general: if the developer's symbol is at least three actual characters and ends with "`}`", it gets prefixed with "`S:`".

In state $q_{\texttt{doShifts}}$:

- On symbol "␣", replace with "1", do not move the tape head, and change to state $q_{\texttt{ACCEPT}}$
- On symbol "0", replace with "␣", move right, replace the symbol there with "0", and then move left two cells, ending in state $q_{\texttt{doShifts}}$
- On symbol "1", replace with "␣", move right, replace the symbol there with "1", and then move left two cells, ending in state $q_{\texttt{doShifts}}$
- On symbol "2", replace with "␣", move right, replace the symbol there with "2", and then move left two cells, ending in state $q_{\texttt{doShifts}}$
- On symbol "3", replace with "␣", move right, replace the symbol there with "3", and then move left two cells, ending in state $q_{\texttt{doShifts}}$
- On symbol "4", replace with "␣", move right, replace the symbol there with "4", and then move left two cells, ending in state $q_{\texttt{doShifts}}$
- On symbol "5", replace with "␣", move right, replace the symbol there with "5", and then move left two cells, ending in state $q_{\texttt{doShifts}}$
- On symbol "6", replace with "␣", move right, replace the symbol there with "6", and then move left two cells, ending in state $q_{\texttt{doShifts}}$
- On symbol "7", replace with "␣", move right, replace the symbol there with "7", and then move left two cells, ending in state $q_{\texttt{doShifts}}$
- On symbol "8", replace with "␣", move right, replace the symbol there with "8", and then move left two cells, ending in state $q_{\texttt{doShifts}}$
- On symbol "9", replace with "␣", move right, replace the symbol there with "9", and then move left two cells, ending in state $q_{\texttt{doShifts}}$

```
TM.getState( "q_START" )
 .addTransition( EMPTY_CELL ,  "1"  , N, STATE_ACCEPT )
 .addTransition( WILDCARD, NO_CHANGE, R,  "q_findEnd" )

TM.getState( "q_findEnd" )
 .addTransition( EMPTY_CELL , EMPTY_CELL , L,  "q_doShifts" )
 .addTransition( WILDCARD, NO_CHANGE, R,  "q_findEnd" )
```

```
TM.getState( "q_doShifts" )
 .addTransition( EMPTY_CELL ,  "1"  , N, STATE_ACCEPT )
 .addTransitionGadget(   "0"  , "SHIFT", L(2), "q_doShifts" )
 .addTransitionGadget(   "1"  , "SHIFT", L(2), "q_doShifts" )
 .addTransitionGadget(   "2"  , "SHIFT", L(2), "q_doShifts" )
 .addTransitionGadget(   "3"  , "SHIFT", L(2), "q_doShifts" )
 .addTransitionGadget(   "4"  , "SHIFT", L(2), "q_doShifts" )
 .addTransitionGadget(   "5"  , "SHIFT", L(2), "q_doShifts" )
 .addTransitionGadget(   "6"  , "SHIFT", L(2), "q_doShifts" )
 .addTransitionGadget(   "7"  , "SHIFT", L(2), "q_doShifts" )
 .addTransitionGadget(   "8"  , "SHIFT", L(2), "q_doShifts" )
 .addTransitionGadget(   "9"  , "SHIFT", L(2), "q_doShifts" )
```

Feature #7: Cell-shifting gadget matching multiple symbols

Clearly, the amount of duplication left can still become problematic. Accordingly, let's allow defining multiple options for the gadget to match against at the same time, the same way we did with transitions:

Program #15: Insert "1" before input (gadget matching multiple)

Given a series of digits, insert an extra "1" before the input, shifting all symbols to the right.

In state $q_{\texttt{START}}$:
- On symbol "␣", replace with "1", do not move the tape head, and change to state $q_{\texttt{ACCEPT}}$
- On any other symbol, leave symbol unchanged, move right, and change to state $q_{\texttt{findEnd}}$

In state $q_{\texttt{findEnd}}$:
- On symbol "␣", replace with "␣", move left, and change to state $q_{\texttt{doShifts}}$
- On any other symbol, leave symbol unchanged, move right, and change to state $q_{\texttt{findEnd}}$

In state $q_{\texttt{doShifts}}$:
- On symbol "␣", replace with "1", do not move the tape head, and change to state $q_{\texttt{ACCEPT}}$
- On any of the symbols ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9"], replace with "␣", move right, replace the symbol there with the originally matched symbol, and then move left two cells, ending in state $q_{\texttt{doShifts}}$

```
TM.getState( "qSTART" )
 .addTransition( EMPTY_CELL ,  "1"  , N, STATE_ACCEPT )
 .addTransition( WILDCARD, NO_CHANGE, R,  "qfindEnd" )

TM.getState( "qfindEnd" )
 .addTransition( EMPTY_CELL , EMPTY_CELL , L,  "qdoShifts" )
 .addTransition( WILDCARD, NO_CHANGE, R,  "qfindEnd" )

TM.getState( "qdoShifts" )
 .addTransition( EMPTY_CELL ,  "1"  , N, STATE_ACCEPT )
 .addTransitionGadget(  ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9"],
    "SHIFT", L(2),  "qdoShifts" )
```

Feature #8: Cell-shifting gadget as fallback

But why stop there? The developer still needs to write out each symbol for which the gadget should be used, and with larger tape alphabets this can get tedious. Additionally, this introduces a risk of a symbol being missed from the list accidentally. Accordingly, let's allow using the gadget to be a fallback transition, the same way we can have the fallback be any other transition:

Program #16: Insert "1" before input (gadget as fallback)

Given a series of digits, insert an extra "1" before the input, shifting all symbols to the right.

In state $q_{\texttt{START}}$:
- On symbol "␣", replace with "1", do not move the tape head, and change to state $q_{\texttt{ACCEPT}}$
- On any other symbol, leave symbol unchanged, move right, and change to state $q_{\texttt{findEnd}}$

In state $q_{\texttt{findEnd}}$:
- On symbol "␣", replace with "␣", move left, and change to state $q_{\texttt{doShifts}}$
- On any other symbol, leave symbol unchanged, move right, and change to state $q_{\texttt{findEnd}}$

In state $q_{\texttt{doShifts}}$:
- On symbol "␣", replace with "1", do not move the tape head, and change to state $q_{\texttt{ACCEPT}}$
- On any other symbol, replace with "␣", move right, replace the symbol there with the originally matched symbol, and then move left two cells, ending in state $q_{\texttt{doShifts}}$

```
TM.getState( "q_START" )
 .addTransition( EMPTY_CELL ,  "1"  , N, STATE_ACCEPT )
 .addTransition( WILDCARD, NO_CHANGE, R,  "q_findEnd" )

TM.getState( "q_findEnd" )
 .addTransition( EMPTY_CELL , EMPTY_CELL , L,  "q_doShifts" )
 .addTransition( WILDCARD, NO_CHANGE, R,  "q_findEnd" )

TM.getState( "q_doShifts" )
 .addTransition( EMPTY_CELL ,  "1"  , N, STATE_ACCEPT )
 .addTransitionGadget(  WILDCARD, "SHIFT", L(2),  "q_doShifts" )
```

At this point, the way that the developer defines transitions is no longer as simple to give a signature for. The developer can still specify transitions using all the previous features, but they can now also add a "gadget" that doesn't cleanly fit into a signature. Accordingly, let's define a new "type" of values for a signature, for use by gadgets: G. A gadget G starts in a state and upon matching some symbol, and represents a series of specific steps specified by the compiler (or potentially supported by the underlying machine). Every gadget must have an end state, and will also have one or more updates to the tape (including writing symbols and moving the head of the tape). The machine still gets normal transitions if the gadget is not available, but the developer doesn't need to write them all out. At the moment the shifting of a cell is the only gadget defined, but later on others will be introduced. Thus, the developer now defines transitions using either of

$$Q \times (\Gamma | \Gamma[] | \texttt{WILDCARD}) \times Q \times (\Gamma | \texttt{NO\_CHANGE}) \times \{L, R, N, L(v), R(v)\}$$
$$Q \times (\Gamma | \Gamma[] | \texttt{WILDCARD}) \times G(Q, ...)$$

where $G(Q, ...)$ indicates a gadget that ends in a state in Q and may have other parameters (the "..."). In the case of shifting cells over, there is one extra parameter - where to move after completing the shift.

Since different gadgets can have different parameters to hold different data, it no longer makes sense to explain them in terms of the abtract function DefTransition, but the explanation of each gadget should make it clear how it is implemented in terms of the prior features.

2 (f) Inserting first cell markers

Our final contrived example is as follows: given a series of digits, accept if the last digit is odd, and reject if it is even (or if the series of digits is empty). But, when the machine accepts or rejects, do so with the head of the machine pointing at the first symbol from the input, i.e. at the start of the number. It

is fairly easy to determine if the machine should accept or reject, but getting back to the first input symbol is not as straightforward.

After going through all of the input to determine if we should accept or reject, how can we ensure we get back to the start? While each symbol could be checked to see if it is the start, that gets extremely tedious.[14] Luckily, we already know how to insert an extra symbol at the start of the input, from the example above of inserting a "1". So that part isn't going to be too tricky after all. For this example, let's designate "$" to be that extra symbol that gets inserted at the start.

If the input is empty, we could reject immediately. However, to simplify the logic, let's just always insert the "$" at the start. Thus, the high-level algorithm for this program would be:

- Insert a "$" at the start, moving all input symbols to the right.
- If there is no input symbol after the "$", reject.
- Otherwise, go to the end of the input, and check the last input symbol:
  - If the end of the input is an odd digit, go back to the "$", move to the right (to the first symbol from the input), and accept.
  - Otherwise, go back to the "$", move to the right, and reject.

As a program, this would look like:

Program #17: Accept odd numbers, head at start of input (manual insertion of "$")
Given a series of digits, accept if the last digit is odd, and reject if it is even or the input is empty. At the end, leave the tape head at the start of the input.
In state $q_{\texttt{START}}$: • On any symbol, leave symbol unchanged, move right, and change to state $q_{\texttt{findEnd}}$ In state $q_{\texttt{findEnd}}$: • On symbol "␣", replace with "␣", move left, and change to state $q_{\texttt{doShifts}}$ • On any other symbol, leave symbol unchanged, move right, and change to state $q_{\texttt{findEnd}}$

[14] For example, by erasing it and then moving left. If after the move the machine is looking at an empty cell, the start of the tape was found; if not, go to the right, replace the symbol that was erased, and then go left to the next cell to test if that is the start of the tape.

In state $q_{\texttt{doShifts}}$:
- On symbol "␣", replace with "$", move right, and change to state $q_{\texttt{checkFirstInput}}$
- On any other symbol, replace with "␣", move right, replace the symbol there with the originally matched symbol, and then move left two cells, ending in state $q_{\texttt{doShifts}}$

In state $q_{\texttt{checkFirstInput}}$:
- On symbol "␣", replace with "␣", do not move the tape head, and change to state $q_{\texttt{REJECT}}$
- On any other symbol, leave symbol unchanged, move right, and change to state $q_{\texttt{findLastInput}}$

In state $q_{\texttt{findLastInput}}$:
- On symbol "␣", replace with "␣", move left, and change to state $q_{\texttt{checkLastInput}}$
- On any other symbol, leave symbol unchanged, move right, and change to state $q_{\texttt{findLastInput}}$

In state $q_{\texttt{checkLastInput}}$:
- On any of the symbols ["1", "3", "5", "7", "9"], leave symbol unchanged, move left, and change to state $q_{\texttt{findFirstInputThenAccept}}$
- On any of the symbols ["2", "4", "6", "8", "0"], leave symbol unchanged, move left, and change to state $q_{\texttt{findFirstInputThenReject}}$

In state $q_{\texttt{findFirstInputThenAccept}}$:
- On symbol "$", replace with "$", move right, and change to state $q_{\texttt{ACCEPT}}$
- On any other symbol, leave symbol unchanged, move left, and change to state $q_{\texttt{findFirstInputThenAccept}}$

In state $q_{\texttt{findFirstInputThenReject}}$:
- On symbol "$", replace with "$", move right, and change to state $q_{\texttt{REJECT}}$
- On any other symbol, leave symbol unchanged, move left, and change to state $q_{\texttt{findFirstInputThenReject}}$

```
TM.getState( "qSTART" )
 .addTransition( WILDCARD, NO_CHANGE, R, "qfindEnd" )

TM.getState( "qfindEnd" )
 .addTransition( EMPTY_CELL , EMPTY_CELL , L, "qdoShifts" )
 .addTransition( WILDCARD, NO_CHANGE, R, "qfindEnd" )
```

```
TM.getState( "q_doShifts" )
 .addTransition( EMPTY_CELL , "$" , R, "q_checkFirstInput" )
 .addTransitionGadget( WILDCARD, "SHIFT", L(2), "q_doShifts" )

TM.getState( "q_checkFirstInput" )
 .addTransition( EMPTY_CELL , EMPTY_CELL , N,
    STATE_REJECT )
 .addTransition( WILDCARD, NO_CHANGE, R, "q_findLastInput" )

TM.getState( "q_findLastInput" )
 .addTransition( EMPTY_CELL , EMPTY_CELL , L, "q_checkLastInput" )
 .addTransition( WILDCARD, NO_CHANGE, R, "q_findLastInput" )

TM.getState( "q_checkLastInput" )
 .addTransition( ["1", "3", "5", "7", "9"], NO_CHANGE, L,
    "q_findFirstInputThenAccept" )
 .addTransition( ["2", "4", "6", "8", "0"], NO_CHANGE, L,
    "q_findFirstInputThenReject" )

TM.getState( "q_findFirstInputThenAccept" )
 .addTransition( "$" , "$" , R, STATE_ACCEPT )
 .addTransition( WILDCARD, NO_CHANGE, L, "q_findFirstInputThenAccept"
    )

TM.getState( "q_findFirstInputThenReject" )
 .addTransition( "$" , "$" , R, STATE_REJECT )
 .addTransition( WILDCARD, NO_CHANGE, L, "q_findFirstInputThenReject"
    )
```

Feature #9: First-cell marker gadget

As it turns out, inserting a marker at the first cell is going to be very helpful for future features.[15] Accordingly, let's add a gadget to insert a specific symbol at the start of the tape. The gadget:

- Expects to be used directly at the start of the program in $q_{\texttt{START}}$ and to be used for all possible target symbols (i.e. with a wildcard).
- Uses the same transitions as $q_{\texttt{findEnd}}$ and $q_{\texttt{doShifts}}$ above, but under names that get reserved by the compiler.

[15] In fact, it could have been used for the prior contrived example of inserting a "1" at the start of the input. But, I figured it would be easier to understand if a separate example program was used for this new feature, since the shifting feature can and will also be used independently of the goal of adding a symbol to the start of the tape.

- Is told what symbol to insert at the start, and what state to end in after inserting the marker.
- Ends in the specified state with the head of the tape pointing to the second cell on the tape, which contains the first input symbol (or is empty if the overall input was empty).

To implement this gadget, the compiler will generate state names in the form

- `ω_firstCellMarker_ω_{start state}_ω_{match content}_ω_findEnd`
- `ω_firstCellMarker_ω_{start state}_ω_{match content}_ω_doShifts`

No other compiler feature should generate such state names. Since the state names generated as based on the state and symbol that the gadget was used it, different uses of the gadget should not conflict.

Using this gadget, the program would be simplified to (note that only the first part has changed, once the "$" is inserted the rest of the logic remains the same):

Program #18: Accept odd numbers, head at start of input (gadget insertion of "$")
Given a series of digits, accept if the last digit is odd, and reject if it is even or the input is empty. At the end, leave the tape head at the start of the input.
In state $q_{\texttt{START}}$: • On any symbol, insert a "$", shifting all input symbols to the right, then move right to the second cell, ending in state $q_{\texttt{checkFirstInput}}$ In state $q_{\texttt{checkFirstInput}}$: • On symbol "␣", replace with "␣", do not move the tape head, and change to state $q_{\texttt{REJECT}}$ • On any other symbol, leave symbol unchanged, move right, and change to state $q_{\texttt{findLastInput}}$ In state $q_{\texttt{findLastInput}}$: • On symbol "␣", replace with "␣", move left, and change to state $q_{\texttt{checkLastInput}}$ • On any other symbol, leave symbol unchanged, move right, and change to state $q_{\texttt{findLastInput}}$

In state $q_{\texttt{checkLastInput}}$:

- On any of the symbols ["1", "3", "5", "7", "9"], leave symbol unchanged, move left, and change to state $q_{\texttt{findFirstInputThenAccept}}$
- On any of the symbols ["2", "4", "6", "8", "0"], leave symbol unchanged, move left, and change to state $q_{\texttt{findFirstInputThenReject}}$

In state $q_{\texttt{findFirstInputThenAccept}}$:

- On symbol "$", replace with "$", move right, and change to state $q_{\textsc{accept}}$
- On any other symbol, leave symbol unchanged, move left, and change to state $q_{\texttt{findFirstInputThenAccept}}$

In state $q_{\texttt{findFirstInputThenReject}}$:

- On symbol "$", replace with "$", move right, and change to state $q_{\textsc{reject}}$
- On any other symbol, leave symbol unchanged, move left, and change to state $q_{\texttt{findFirstInputThenReject}}$

```
TM.getState( "q_START" )
 .addTransitionGadget( WILDCARD, "FIRST-CELL-MARKER", "$" ,
    "q_checkFirstInput" )

TM.getState( "q_checkFirstInput" )
 .addTransition( EMPTY_CELL , EMPTY_CELL , N,
    STATE_REJECT )
 .addTransition( WILDCARD, NO_CHANGE, R, "q_findLastInput" )

TM.getState( "q_findLastInput" )
 .addTransition( EMPTY_CELL , EMPTY_CELL , L, "q_checkLastInput" )
 .addTransition( WILDCARD, NO_CHANGE, R, "q_findLastInput" )

TM.getState( "q_checkLastInput" )
 .addTransition( ["1", "3", "5", "7", "9"], NO_CHANGE, L,
    "q_findFirstInputThenAccept" )
 .addTransition( ["2", "4", "6", "8", "0"], NO_CHANGE, L,
    "q_findFirstInputThenReject" )

TM.getState( "q_findFirstInputThenAccept" )
 .addTransition( "$" , "$" , R, STATE_ACCEPT )
 .addTransition( WILDCARD, NO_CHANGE, L, "q_findFirstInputThenAccept"
    )
```

```
TM.getState( "q_findFirstInputThenReject" )
 .addTransition(  "$" ,  "$" , R, STATE_REJECT )
 .addTransition( WILDCARD, NO_CHANGE, L, "q_findFirstInputThenReject"
    )
```

At this point the way that the developer defines transitions remains

$$Q \times (\Gamma|\Gamma[]|\texttt{WILDCARD}) \times Q \times (\Gamma|\texttt{NO\_CHANGE}) \times \{L, R, N, L(v), R(v)\}$$
$$Q \times (\Gamma|\Gamma[]|\texttt{WILDCARD}) \times G(Q, ...)$$

but the available gadgets are

1. Shift the current cell's symbol to the right (extra parameter: movement to make after the shift).
2. Insert a specific symbol, shifting all symbols to the right (extra parameter: the symbol to insert).

2 (g) Putting it all together

We now arrive at an example program that isn't as contrived: Given a series of digits, determine if the number of digits (*not* the number that the digits make up) in the input is a power of two, accepting if it is and rejecting if it isn't. We could easily change this program to instead use a different input alphabet (say, all of the lowercase letters, or a combination of letters and numbers), the actual alphabet is not important. This is merely a demonstration of a semi-useful program that makes use of the various features introduced above.

Determining if a positive whole number n (which must be greater than "0") is a power of two can be done as follows:

- Cross out every other digit.
  - if only 1 digit was found, n is a power of 2 and accept.
  - if an odd number of digits were found, n is not a power of 2 and reject.
  - if an even number of digits were found, check $\frac{n}{2}$ by running again.

However, translating this into states and transitions, even with our extra features and gadgets, is not straightforward. After going through all of the input the first time, how can we ensure we get back to the start? Luckily, we just added a gadget to insert a marker for the first cell.

Another issue is how to ensure the entire input is processed on subsequent passes through the symbols. If we "cross out" each digit by replacing it with "␣" we will have no way to determine if a given "␣" is at the end of the input or in the middle! Accordingly, let's add a symbol, in this case the letter "x",

that is *not* in the input alphabet, to the tape alphabet.[16] Accordingly, our updated algorithm is:

- Insert a "$" before the input symbols.
- Replace every other digit with an "x", skipping past any existing "x".
  - if only 1 digit was found, n is a power of 2 and accept.
  - if an odd number of digits were found, n is not a power of 2 and reject.
  - if an even number of digits were found, we need to check $\frac{n}{2}$.
- Check $\frac{n}{2}$ by searching for the "$", and repeat the processing from the previous step.

With the specific states and transitions (making use of our gadgets and extra features) this would look like the following:

Program #19: Check power of two

Given a series of digits, accept if the number of digits is a power of two, reject otherwise.

In state $q_{\texttt{START}}$:
- On any symbol, insert a "$", shifting all input symbols to the right, then move right to the second cell, ending in state $q_{\texttt{found0Digits}}$

In state $q_{\texttt{found0Digits}}$:
- On symbol "x", replace with "x", move right, and change to state $q_{\texttt{found0Digits}}$
- On symbol "␣", replace with "␣", do not move the tape head, and change to state $q_{\texttt{REJECT}}$
- On any other symbol, replace with "x", move right, and change to state $q_{\texttt{found1Digit}}$

In state $q_{\texttt{found1Digit}}$:
- On symbol "x", replace with "x", move right, and change to state $q_{\texttt{found1Digit}}$
- On symbol "␣", replace with "␣", do not move the tape head, and change to state $q_{\texttt{ACCEPT}}$
- On any other symbol, leave symbol unchanged, move right, and change to state $q_{\texttt{foundEvenDigits}}$

[16] If the input alphabet includes an "x", pick another letter, or invent an entirely new symbol, to use instead. The only requirement is that the marker symbol not be a valid input symbol (i.e. not included in the set Σ) or be the empty symbol.

In state $q_{\texttt{foundEvenDigits}}$:

- On symbol "x", replace with "x", move right, and change to state $q_{\texttt{foundEvenDigits}}$
- On symbol "␣", replace with "␣", move left, and change to state $q_{\texttt{returnToStart}}$
- On any other symbol, replace with "x", move right, and change to state $q_{\texttt{foundOddDigits}}$

In state $q_{\texttt{returnToStart}}$:

- On symbol "$", replace with "$", move right, and change to state $q_{\texttt{found0Digits}}$
- On any other symbol, leave symbol unchanged, move left, and change to state $q_{\texttt{returnToStart}}$

In state $q_{\texttt{foundOddDigits}}$:

- On symbol "x", replace with "x", move right, and change to state $q_{\texttt{foundOddDigits}}$
- On symbol "␣", replace with "␣", do not move the tape head, and change to state $q_{\texttt{REJECT}}$
- On any other symbol, leave symbol unchanged, move right, and change to state $q_{\texttt{foundEvenDigits}}$

```
TM.getState( "q_START" )
 .addTransitionGadget( WILDCARD, "FIRST-CELL-MARKER", "$" ,
    "q_found0Digits" )

TM.getState( "q_found0Digits" )
 .addTransition( "x" , "x" , R, "q_found0Digits" )
 .addTransition( EMPTY_CELL , EMPTY_CELL , N,
    STATE_REJECT )
 .addTransition( WILDCARD, "x" , R, "q_found1Digit" )

TM.getState( "q_found1Digit" )
 .addTransition( "x" , "x" , R, "q_found1Digit" )
 .addTransition( EMPTY_CELL , EMPTY_CELL , N,
    STATE_ACCEPT )
 .addTransition( WILDCARD, NO_CHANGE, R, "q_foundEvenDigits" )

TM.getState( "q_foundEvenDigits" )
 .addTransition( "x" , "x" , R, "q_foundEvenDigits" )
 .addTransition( EMPTY_CELL , EMPTY_CELL , L, "q_returnToStart" )
 .addTransition( WILDCARD, "x" , R, "q_foundOddDigits" )

TM.getState( "q_returnToStart" )
 .addTransition( "$" , "$" , R, "q_found0Digits" )
 .addTransition( WILDCARD, NO_CHANGE, L, "q_returnToStart" )
```

```
TM.getState( "q_foundOddDigits" )
 .addTransition( "x" , "x" , R, "q_foundOddDigits" )
 .addTransition( EMPTY_CELL , EMPTY_CELL , N,
    STATE_REJECT )
 .addTransition( WILDCARD, NO_CHANGE, R, "q_foundEvenDigits" )
```

The program crosses out every other digit by alternating between states that cross out the digits and states that simply skip past digits. Step by step records of the program executing for the inputs `"123456"` (rejected for having six digits) and `"7890"` (accepted for having four digits) are included in an appendix.

2 (h) Inserting a cell after the start of the tape

Before we get into how to simulate a machine that runs on multiple tapes, there is another gadget that would be useful:

Once we have a way to make the start of a tape, it will be helpful to be able to add a new cell directly after the start symbol, leaving the start symbol in place but shifting everything else over by one cell. If the developer had to define such a feature manually, it might look as follows:

- Insert a `"$"` at the start, moving all input symbols to the right.
- Following the same procedure as the insertion of `"$"`, shift all cells over by one starting at the end.
- ...BUT instead of also shifting over the `"$"`, when it is reached:
  - move right to the empty cell, and
  - insert the desired cell value.

As a program, this would look like (if we want to insert a `"$"` and then a `"1"` at the start of a series of `"0"`s and `"1"`s):

Program #20: Insert "$" and "1" at start (manual insertion of "1")
Given a series of `"0"`s and `"1"`s, insert a `"$"` and then a `"1"` at the start.
In state $q_{\texttt{START}}$: • On any symbol, insert a "$", shifting all input symbols to the right, then move right to the second cell, ending in state $q_{\texttt{findEnd}}$

In state $q_{\texttt{findEnd}}$:
- On symbol "␣", replace with "␣", move left, and change to state $q_{\texttt{doShifts}}$
- On any other symbol, leave symbol unchanged, move right, and change to state $q_{\texttt{findEnd}}$

In state $q_{\texttt{doShifts}}$:
- On symbol "$", replace with "$", move right, and change to state $q_{\texttt{doInsert1}}$
- On any other symbol, replace with "␣", move right, replace the symbol there with the originally matched symbol, and then move left two cells, ending in state $q_{\texttt{doShifts}}$

In state $q_{\texttt{doInsert1}}$:
- On symbol "␣", replace with "1", move right, and change to state $q_{\textsc{accept}}$

```
TM.getState( "q_START" )
 .addTransitionGadget( WILDCARD, "FIRST-CELL-MARKER", "$" ,
    "q_findEnd" )

TM.getState( "q_findEnd" )
 .addTransition( EMPTY_CELL , EMPTY_CELL , L, "q_doShifts" )
 .addTransition( WILDCARD, NO_CHANGE, R, "q_findEnd" )

TM.getState( "q_doShifts" )
 .addTransition( "$" , "$" , R, "q_doInsert1" )
 .addTransitionGadget( WILDCARD, "SHIFT", L(2), "q_doShifts" )

TM.getState( "q_doInsert1" )
 .addTransition( EMPTY_CELL , "1" , R, STATE_ACCEPT )
```

Feature #10: After-marker insertion gadget

If we were to add a gadget to do this, it would instead look like:

Program #21: Insert "$" and "1" at start (gadget insertion of "1")

Given a series of "0"s and "1"s, insert a "$" and then a "1" at the start.

In state $q_{\textsc{start}}$:
- On any symbol, insert a "$", shifting all input symbols to the right, then move right to the second cell, ending in state $q_{\texttt{insert1}}$

In state $q_{\texttt{insert1}}$:
- On any symbol, insert a "1" after a "$", then move right to the next cell, ending in state $q_{\textsc{accept}}$

```
TM.getState( "q_START" )
 .addTransitionGadget(  WILDCARD, "FIRST-CELL-MARKER",   "$"  ,
    "q_insert1" )

TM.getState( "q_insert1" )
 .addTransitionGadget(  WILDCARD, "INSERT-AFTER-MARKER",  " $" ,
    "1"  , STATE_ACCEPT )
```

To implement this gadget, the compiler will generate state names in the forms

- `ω_afterCellMarker_ω_{target state}_ω_{match content}_findEnd`
- `ω_afterCellMarker_ω_{target state}_ω_{match content}_doShifts`
- `ω_afterCellMarker_ω_{target state}_ω_{match content}_doInsert`

No other compiler feature should generate such state names. Since the state names generated as based on the state and symbol that the gadget was used it, different uses of the gadget should not conflict.

The list of available gadgets is now:

1. Shift the current cell's symbol to the right (extra parameter: movement to make after the shift).
2. Insert a specific symbol, shifting all symbols to the right (extra parameter: the symbol to insert).
3. Insert a symbol after some specified marker, shifting subsequent symbols to the right (extra parameters: the symbol to find and the symbol to insert after it).

Using these, we are now ready to work on simulating multi-tape machines.

3 Using multi-tape machines

For the examples above, I first showed the version of a program without a specific feature before introducing that feature. However, in the case of multiple tapes being supported on a single-taped underlying machine, the single-taped version is much more complicated and harder to understand than the eventual version using the multiple-tapes feature. While this was true to a limited extent with all of the features previously introduced, the simulation of multiple tapes is much more complicated than simulating moving multiple cells or the other previous features. Accordingly, in this case I'll walk through using the feature from the developer's perspective first, and afterwards explain how multiple tapes could be implemented.

3 (a) Example program

In the contrived example for using multiple tapes (starting with two tapes), let us imagine an input that is a series of `"0"`s and `"1"`s. The machine should accept if there is the same number of each digit, and reject otherwise, but unlike in our program for determining if there is a power-of-two number of inputs, we should *not* cross out or modify any of the symbols on the tape. Yes, we could cross them out and then revert that later, but the point of this program is to use multiple tapes.

As we process the input digits, we will need to keep track of the difference in the number of `"0"`s and `"1"`s already processed from the input. This overall difference *cannot* be stored in the name of the state the way that we have stored extra information previously, because the set of states must be finalized before the user input is actually processed. Since we need to support arbitrary inputs that could have a difference of any arbitrarily-large magnitude at some point in the middle of the processing, if the difference in the `"0"`s and `"1"`s were stored in the name of the state, there would always be some inputs that would need more states than were configured.[17] However, we can store whether this difference is positive or negative (i.e. if there have been more `"0"`s or `"1"`s) in the state name, because there are a finite number of options possible there (two). For the actual magnitude of the difference, let's use a second tape - the extent to which the head of the second tape has moved right indicates how many more `"0"`s or `"1"`s have been processed.

For the tape with the input symbols, we don't need to insert a first cell marker, since we do not need to be able to go back to the start of the input. But, for the tape that keeps track of the magnitude of the difference, we do need such a marker - that is how we know if the symbol that we have found more of switches! Let us use `"$"` as that marker. Accordingly, the program might look something like the following.

[17] For example, if $|Q|$ is the number of states in Q, an input of $|Q| + 1$ `"0"`s followed by the same number of `"1"`s could not be supported properly.

- In the start state:
  - If the input tape cell is empty, meaning that the overall input was empty, write an empty cell on the input tape, a "$" on the tracking tape, and accept.
  - If the input tape cell is a "0", write a "0" on the input tape cell, a "$" on the tracking tape cell, move right on both tapes, and change to a state indicating more "0"s than "1"s.
  - If the input tape cell is a "1", write a "1" on the input tape cell, a "$" on the tracking tape cell, move right on both tapes, and change to a state indicating more "1"s than "0"s.
- In the state when there have been at least as many "0"s as "1"s (#"0"s >= #"1"s):
  - If the input tape cell is empty, accept if the head of the tracking tape is at the cell that holds the "$", otherwise reject.
  - If the input tape cell is a "0", moving the head of the tracking tape to the right.
  - If the input tape cell is a "1", if the tracking tape cell holds the "$" move that tape right and change to the state with more "1"s, otherwise move that tracking tape head left.
  - On all of the transitions above, the head of the input tape moves right.
- In the state where there have been at least as many "1"s as "0"s, just do the symmetric operation.

When there has been an equal number of "0"s and "1"s processed, the machine may be in either of the two states for tracking the difference, depending on what the last digit processed was. Regardless of which state it is in, if an equal number of "0"s and "1"s have been processed, the head of the tracking tape will always be over the "$".

For now, we assume that none of the features added previously are available (yet) - no matching multiple options or a wildcard, no moving over multiple cells at once, and no gadgets. **But**, we do require that when using multiple real tapes it be possible for a transition to specify that the head of a tape (or the heads of multiple tapes) not move at all. If the machine offers multiple real tapes but lacks the feature to not move the head of the tape, the compiler will only make use of the first real tape and use it to simulate the desired extra tapes.[18] Thus, for multi-tape transitions, the type of the actual transition function will be

[18]Support for no-move transitions is needed so that the relative positions of tape heads can be changed properly, for example if the head of tape 0 is at the index exactly one cell to the right of the head of tape 1, no amount of pure left and right movements would get the two

$$Q \times (\Gamma)^n \longrightarrow Q \times (\Gamma)^n \times (\{L, R, N\})^n$$

where n is the number of tapes. The developer will define the transitions using tuples in the form:

$$Q \times (\Gamma)^n \times Q \times (\Gamma)^n \times (\{L, R, N\})^n$$

Symbols and movements in the form <a, b> indicate that the first ("a") applies to the first tape, and the second ("b") applies to the second tape, and so on for sets with more elements; the number of elements must be consistent with the number of tapes the machine is using.

Program #22: Require equal "0"s and "1"s (two tapes)
Given a series of "0"s and "1"s, accept if there is an equal number of each digit, reject otherwise.
In state $q_{\texttt{Run}}$: • When the tapes have symbols <"␣", "␣">, replace them with <"␣", "$">, then <do not move the tape head, do not move the tape head>, and change to state $q_{\textsc{Accept}}$ • When the tapes have symbols <"0", "␣">, replace them with <"0", "$">, then <move right, move right>, and change to state $q_{\texttt{more0s}}$ • When the tapes have symbols <"1", "␣">, replace them with <"1", "$">, then <move right, move right>, and change to state $q_{\texttt{more1s}}$

to align (be at the same index) without going all the way back to the left edge of the tape, where a left movement functions as a no-move transition. This could be worked around by only using every other cell, for example, but since we need to support simulating extra tapes, its easier to just say that if the real extra tapes do not support no-move transitions, use the simulation, where we already know that the compiler can provide the no-move transitions.

In state $q_{\texttt{more0s}}$:

- When the tapes have symbols <"␣", "$">, replace them with <"␣", "$">, then <do not move the tape head, do not move the tape head>, and change to state $q_{\textsc{accept}}$
- When the tapes have symbols <"␣", "␣">, replace them with <"␣", "␣">, then <do not move the tape head, do not move the tape head>, and change to state $q_{\textsc{reject}}$
- When the tapes have symbols <"0", "␣">, replace them with <"0", "␣">, then <move right, move right>, and change to state $q_{\texttt{more0s}}$
- When the tapes have symbols <"0", "$">, replace them with <"0", "$">, then <move right, move right>, and change to state $q_{\texttt{more0s}}$
- When the tapes have symbols <"1", "␣">, replace them with <"1", "␣">, then <move right, move left>, and change to state $q_{\texttt{more0s}}$
- When the tapes have symbols <"1", "$">, replace them with <"1", "$">, then <move right, move right>, and change to state $q_{\texttt{more1s}}$

In state $q_{\texttt{more1s}}$:

- When the tapes have symbols <"␣", "$">, replace them with <"␣", "$">, then <do not move the tape head, do not move the tape head>, and change to state $q_{\textsc{accept}}$
- When the tapes have symbols <"␣", "␣">, replace them with <"␣", "␣">, then <do not move the tape head, do not move the tape head>, and change to state $q_{\textsc{reject}}$
- When the tapes have symbols <"1", "␣">, replace them with <"1", "␣">, then <move right, move right>, and change to state $q_{\texttt{more1s}}$
- When the tapes have symbols <"1", "$">, replace them with <"1", "$">, then <move right, move right>, and change to state $q_{\texttt{more1s}}$
- When the tapes have symbols <"0", "␣">, replace them with <"0", "␣">, then <move right, move left>, and change to state $q_{\texttt{more1s}}$
- When the tapes have symbols <"0", "$">, replace them with <"0", "$">, then <move right, move right>, and change to state $q_{\texttt{more0s}}$

```
TM.getState( "q_Run" )
 .addMTapeTransition( [EMPTY_CELL , EMPTY_CELL ],
    [EMPTY_CELL , "$" ], [N, N], STATE_ACCEPT )
 .addMTapeTransition( [ "0" , EMPTY_CELL ], [ "0" , "$" ], [R, R],
    "q_more0s" )
 .addMTapeTransition( [ "1" , EMPTY_CELL ], [ "1" , "$" ], [R, R],
    "q_more1s" )
```

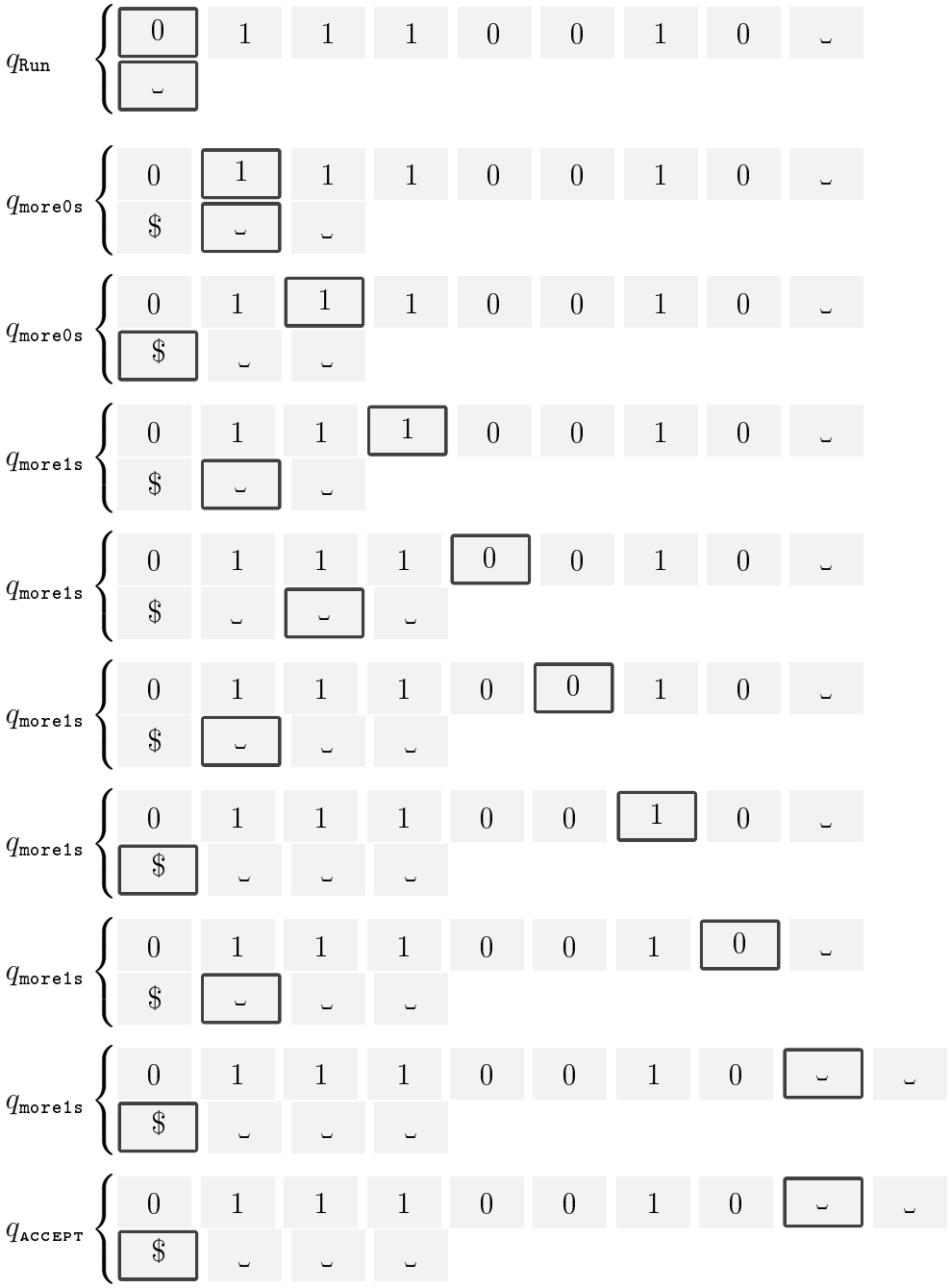
```
TM.getState( "q_more0s" )
 .addMTapeTransition( [EMPTY_CELL , "$" ], [EMPTY_CELL , "$"
    ], [N, N], STATE_ACCEPT )
 .addMTapeTransition( [EMPTY_CELL , EMPTY_CELL ],
    [EMPTY_CELL , EMPTY_CELL ], [N, N], STATE_REJECT
    )
 .addMTapeTransition( [ "0" , EMPTY_CELL ], [ "0" , EMPTY_CELL
    ], [R, R], "q_more0s" )
 .addMTapeTransition( [ "0" , "$" ], [ "0" , "$" ], [R, R], "q_more0s" )
 .addMTapeTransition( [ "1" , EMPTY_CELL ], [ "1" , EMPTY_CELL
    ], [R, L], "q_more0s" )
 .addMTapeTransition( [ "1" , "$" ], [ "1" , "$" ], [R, R], "q_more1s" )

TM.getState( "q_more1s" )
 .addMTapeTransition( [EMPTY_CELL , "$" ], [EMPTY_CELL , "$"
    ], [N, N], STATE_ACCEPT )
 .addMTapeTransition( [EMPTY_CELL , EMPTY_CELL ],
    [EMPTY_CELL , EMPTY_CELL ], [N, N], STATE_REJECT
    )
 .addMTapeTransition( [ "1" , EMPTY_CELL ], [ "1" , EMPTY_CELL
    ], [R, R], "q_more1s" )
 .addMTapeTransition( [ "1" , "$" ], [ "1" , "$" ], [R, R], "q_more1s" )
 .addMTapeTransition( [ "0" , EMPTY_CELL ], [ "0" , EMPTY_CELL
    ], [R, L], "q_more1s" )
 .addMTapeTransition( [ "0" , "$" ], [ "0" , "$" ], [R, R], "q_more0s" )
```

Observant readers will note that, unlike past examples, the above program starts its actual processing in the state $q_{\texttt{Run}}$ rather than $q_{\texttt{START}}$. This is done in preparation for introducing the process by which a multi-tape machine can be simulated by a single-tape machine. If the single-tape machine begins in the state $q_{\texttt{START}}$, it would be confusing for that state to hold both the single-tape transitions used to set up the multi-tape simulation, and the multi-tape transitions that get simulated. Accordingly, for our programs that use multiple tapes (regardless of whether extra tapes are real or simulated) we will have the actual program begin in a state other than $q_{\texttt{START}}$, so that the $q_{\texttt{START}}$ state can be used for setting up the multi-tape environment if needed.[19]

The execution of this would be as follows, for the input "01110010":

[19] In the code, this is implemented by calling the "setNumTapes()" method on the "TM", specifying the number of tapes to use and what state to begin execution from.



3 (b) Simulating extra tapes

Now that we have seen how a program might work on a multi-tape machine, we return to the question of how to simulate such behavior on a single-tape machine. The first thing that stands out as a difference between a single-taped Turing Machine and one with multiple tapes is that the multi-taped version has multiple tape cells that the machine is pointing to with the tape "heads" at the same time. Specifically, there is a tape "head" for each tape,

and thus there will always be more than just the single tape head that the underlying single-taped TM makes available. Accordingly, we will need some way to identify a tape cell as being the current position of the tape head for its tape, when all of the extra tapes are being simulated.

Since Turing Machines deal purely with the world of "symbols" rather than being restricted to the Latin alphabet or ASCII characters, we can simply have the compiler invent some symbol for each symbol in the developer's tape alphabet that indicates a specific symbol is also the current position of its tape's head. Let these symbols be represented with a bar over the original, i.e. "$\bar{\texttt{x}}$" means an "x" cell that is also the cell currently under the head of its tape. The bar representation here is just for documentation purposes - if the developer wanted to write a program that used symbols that included bars over them, the compiler would just pick some other corresponding symbol. The only requirement is that the generated symbols correspond one-to-one with the original tape alphabet symbols, and none of the generated symbols are included in the original tape alphabet.

Since the compiler will add these extra symbols to the tape alphabet *before* the actual execution of the developer's program, the eventual compiled output has a fixed alphabet, as required by the definition of a Turing Machine.

If multiple tapes are represented on a single tape, the different simulated tapes can either be stored in parallel (alternating a symbol from each tape) or in series (putting all of one simulated tape and then all of the next). Each approach has its own benefits and drawbacks; in an earlier version of my code I indeed implemented the tapes in parallel. However, since it is easier to explain, show, and implement tapes stored in series, I decided to explain how to simulate extra tapes by storing them in series. Thus, we will need some way of delineating the different tapes. Let the compiler invent two new symbols, one to indicate the left end of a simulated tape, and one to indicate the right end. For documentation purposes here I will represent these as "{<}" and "{>}", but again if the developer wants to use symbols that look like these, the compiler will invent some other distinct markers. These and the other symbols that are used to implement simulated tapes are also listed in an appendix at the end of the paper.

At this point, we can now represent the starting state

q_{START}

0	1	1	1	0	0	1	0	␣
␣	␣							

on a single tape as

{<}	$\overline{0}$	1	1	1	0	0	1	0	␣	{>}
{<}	$\overline{\text{␣}}$	␣	{>}							

Our general approach to simulating the transitions needed will be to:

1. Go to the left end of the first tape.
2. Scan right until the first simulated tape head is found, leaving the tape unmodified but recording the cell value in the state name (as a substitute for memory).
3. Scan right until the second tape head is found, again leaving the tape unmodified but recording the cell value in the state name.
4. Based only on the new state name (which contains the details of the starting state and the symbols at the heads of the different tapes), determine the updates needed to the tapes, and apply them from right to left, working back to the start.
5. After applying the update to the first tape, change to the new state and repeat.

When the head of a simulated tape is meant to shift left and it is already at the start of the tape, it should not move - that is fairly simple to implement, instead of marking a "{<}" as the head of the tape, just move right and mark that symbol.[20] However, what if we need to move right and are at the end of a simulated tape? The developer expects the tape to be unbounded on the right, so we need to insert a new empty cell, shifting all subsequent cells over by one. Thus, we need to know which is the first empty cell that can safely be shifted over. While we could grab the empty cell[21] from some other tape where it is directly preceding the end of the tape (i.e. to the left of the "{>}") and not marked as the head of the tape, to make things easier let's just grab the first empty cell to the right of the last simulated tape. And to make things even easier, we will add a special marker after the last tape, so that we don't need to keep track of the number of tapes to cross. Let the compiler invent some new symbol, here represented as "{$$}", to be used.

While we could do the insertion of the empty symbols as part of the application of the rest of the transition (writing the new symbols and moving the simulated tape heads), that would require each compiler-generated state for handling transitions to also have the necessary handling for inserting the empty cells, which would dramatically increase the number of generated states and

[20] If a left shift ended up on "{<}", the head of the simulated tape was already (before the shift) at the leftmost cell of the tape, and should stay there, meaning that the cell to the right of the "{<}" should be marked as the head of the simulated tape. So, move the real head of the tape one cell to the right, onto the first cell of the simulated tape, and mark that cell as the head of the simulated tape.

[21] By "grab" an empty cell, I mean shift a series of cells each one place to the right, up to when the symbol in the cell to the left of the originally empty cell gets moved into that empty cell. Each "shift" of a cell to the right can also be considered a shift of an empty cell to the left, moving along the tape until it gets to where we want it.

transitions. Instead, let's apply all of the empty cell insertion handling once, after all of the tapes have had their new symbols written and the simulated tape heads moved.

We also want some new marker for the start of the real tape, since the "$" which we have been using to indicate the start of the tape might be used by the developer. We want this new symbol to be invented by the compiler and completely hidden from the developer - here I'll represent it as "{^^}".

To be clear, when multiple tapes are being simulated using the compiler (or when there are actually multiple tapes) the gadgets that deal with the "{^^}" are entirely concealed from the developer. As far as the developer is concerned, the tape has multiple tapes, and the symbol "{^^}" does not exist. However, if the underlying machine only has a single tape, the compiler will *create* this symbol, which is not part of the developer's input or tape alphabets, and use it to mark the start of the single tape that simulates multiple tapes. In the examples below that show the "{^^}", the states and transitions shown are what the compiler internally produces, *not* what a developer would write. The developer simply writes the multi-tape program.

Thus, the start of our execution of the program above would be:

{^^}	{<}	$\overline{0}$	1	1	1	0	0	1	0	␣
{>}	{<}	$\overline{␣}$	␣	{>}	{$$}					

and our updated workflow would be:[22]

1. Go to the left end of the first tape.
2. Scan right until the first simulated tape head is found, leaving the tape unmodified but recording the cell value in the state name (as a substitute for memory).
3. Scan right until the second tape head is found, again leaving the tape unmodified but recording the cell value in the state name.
4. Based only on the new state name (which contains the details of the starting state and the symbols at the heads of the different tapes), determine the updates needed to the tapes, and apply them from right to left, working back to the start:
   (a) To apply an update, replace the marked head cell with the developer's new symbol (not marked as the head), and then move left or right (or not at all) based on the developer-specified transition.
   (b) If the tape moved left onto a "{<}", move right one cell.

[22] As I build up the setup procedure, new parts and changes are underlined, to make it clear what is copied from each prior iteration of the procedure versus what is being added.

(c) Mark the current cell as the simulated head of the tape (even if it is a `"{>}"`).

(d) Move left to the previous simulated tape.

5. After applying the update to the first tape, remembering the new target state, go through the simulated tapes and insert extra empty cells where needed (which is any time a `"{>}"` is marked as the head of a tape; each occurrence gets replaced by an empty cell marked as the head of the tape and then a `"{>}"` no longer marked as the head of the tape), ending by switching to the new state.

6. Repeat.

Here again, there would still be some duplication - each (developer-specified) simulated state would need its own handling for inserting the empty cells needed, and that would still bloat the number of generated states and transitions. Instead, let us store the target state after the updates in a specific cell - the one right after the `"{^^}"`, before the first simulated tape.[23] Then, we just need to have the logic to update the empty tape cells a single time, and transition to and from the target state based on the cell value. The workflow would then be:

1. Go to the left end of the first tape.

2. Scan right until the first simulated tape head is found, leaving the tape unmodified but recording the cell value in the state name (as a substitute for memory).

3. Scan right until the second tape head is found, again leaving the tape unmodified but recording the cell value in the state name.

4. Based only on the new state name (which contains the details of the starting state and the symbols at the heads of the different tapes), determine the updates needed to the tapes, and apply them from right to left, working back to the start:

   (a) To apply an update, replace the marked head cell with the developer's new symbol (not marked as the head), and then move left or right (or not at all) based on the developer-specified transition.

   (b) If the tape moved left onto a `"{<}"`, move right one cell.

[23] To be clear, this introduces a new tape symbol (i.e. value in Γ) for each state that a transition can switch to. The compiler is responsible for ensuring that these new symbols cannot conflict with the existing symbols that the developer has access to. In my implementation, these extra state symbols are rendered as `"Q:{start}"`, for example; as noted in footnote n. 13 on page 29, my implementation prevents any developer symbols from rendering like that.

(c) Mark the current cell as the simulated head of the tape (even if it is a "{>}").

(d) Move left to the previous simulated tape.

5. Upon reaching the start of the overall tape, record the target state in the second cell, and switch to a helper state for handling the extra empty cells for each "{>̄}".

6. Go through the simulated tapes and insert extra empty cells where needed (which is any time a "{>}" is marked as the head of a tape; each occurrence gets replaced by an empty cell marked as the head of the tape and then a "{>}" no longer marked as the head of the tape).

7. Switch to the state specified in the second cell of the real tape.

8. Repeat.

All of this is of course after the initial input of symbols is transformed into the format expected by the multi-tape simulation logic. Another point of note is that now that the compiled states no longer correspond specifically to the developer-specified states and transitions, the developer-specified names should not be used. Instead, let us use compiler-generated state names in the form `ωs_{state name}ω` to indicate that `state name` is being simulated, with information about the current head cell values being stored in the form `ωs_{state name}_ω{info}`, where the second `ω_` is used to avoid conflicts with developer-specified state names that happen to look like they have the information the compiler uses.

Within the actual simulation of the extra tapes, a number of additional compiler-generated states are used; the forms of these state names are documented in the appendix on reserved state names.

And with that, we are now able to have tho compiler provide support for simulating extra tapes even when the underlying machine only has a single tape. The result of the compiler's expansion of the two-tape program into a one-tape program is included in an appendix at the end of the paper.

3 (c) Adding our extra features

As I noted above, to begin with we assume that the additional tapes have none of the extra features we previously added for single-tape machines, except for the requirement that no-move transitions be supported. However, for the same reason that specifying a transition for multiple symbols at once, or a transition that moved over multiple cells, was useful on a single-tape machine, it will be useful on a multi-tape machine. Accordingly, let us add some of these features to the multi-tape version.[24] Since by now it should be clear how these can

[24]But, the "gadgets" are *not* added to the multi-tape version. That is not to say that they cannot be ported over easily - they could be implemented by having one tape use a

actually be implemented, I won't spend the time stepping through them in as detailed a manner as I did for the single-tape version of the features.

Feature #11: Multi-match and no-change (multi-tape)

A transition specified to target multiple potential symbols on a specific tape is equivalent to defining a series of individual transitions, one for each of those potential symbols. When a transition is specified to target multiple potential symbols on different tapes, all combinations of targeted symbols are matched. For example, if a transition is specified for tape 0 having one of `"A"`, `"B"`, or `"C"`, and tape 1 having either `"Y"` or `"Z"`, the transition would apply to the six combinations of symbols: <`"A"`, `"Y"`>, <`"A"`, `"Z"`>, <`"B"`, `"Y"`>, <`"B"`, `"Z"`>, <`"C"`, `"Y"`>, and <`"C"`, `"Z"`>.

A no-change update on a tape works the exact same on a multi-tape machine as it does on a single-tape machine. Since the updates are specific to each tape, a transition can be specified that changes the symbol written on one tape but leaves another tape unchanged. The developer now specifies transitions as follows:

$$Q \times (\Gamma|\Gamma[])^n \times Q \times (\Gamma|\texttt{NO\_CHANGE})^n \times (\{L, R, N\})^n$$

Neither of these features (multi-match and no-change) results in a change to the type of the actual transition function, just the type of the tuple that the developer uses to define the transitions.

Feature #12: Wildcard (multi-tape)

Wildcards for individual tapes work similarly to multi-matches. Like in the single-tape version, after replacing the wildcard with the set of all tape symbols, the resulting multi-match is filtered by what transitions are already defined. Thus, the developer can define multi-tape transitions with:

$$Q \times (\Gamma|\Gamma[]|\texttt{WILDCARD})^n \times Q \times (\Gamma|\texttt{NO\_CHANGE})^n \times (\{L, R, N\})^n$$

This is equivalent to defining a transition where each tape's target symbols are the overall tape alphabet, except that if there are any existing transitions for a state they simply get skipped instead of causing an issue with defining multiple transitions for the same inputs.

However, for multi-tape machines it is also useful to specify an overarching wildcard fallback transition, for any combination of symbols on any of the tapes that does not already have a transition defined. This is just a shortcut to specifying a transition with a wildcard for each of the tapes.

---

gadget and the others all have wildcard, no-change, no-move transitions. But, it did not seem necessary to add the gadgets to the multi-tape version of the machine in order to implement Brainfuck, my eventual goal.

$$Q \times \texttt{WILDCARD} \times Q \times (\Gamma|\texttt{NO\_CHANGE})^n \times (\{L, R, N\})^n$$

These two signatures can be combined as:

$$Q \times ((\Gamma|\Gamma[]|\texttt{WILDCARD})^n|\texttt{WILDCARD}) \times Q \times (\Gamma|\texttt{NO\_CHANGE})^n \times (\{L, R, N\})^n$$

It should be noted that, when simulating the additional tapes, even if the underlying machine supports specifying a "default" or "fallback" transition, since the machine examines the simulated heads of the tapes one at a time a general `"WILDCARD"` transition will be stored the same way as a $\texttt{WILDCARD}^n$ transition, i.e. it will be expanded out to the actual various individual combinations of symbols that are targeted.

I will also note that, if a machine supports multiple real tapes but not a default transition, defining a general `"WILDCARD"` transition may cause performance issues. Since a `"WILDCARD"` transition ensures that there will be a transition defined for every combination of target symbols possible, the transition function will store $|\Gamma|^n$ different transitions for the given state, where $|\Gamma|$ is the number of symbols in Γ and n is the number of tapes used. For example, the implementation of Brainfuck that I introduce below uses five tapes and 28 different symbols. For such a machine, a `"WILDCARD"` transition would mean that the state has $28^5 = 17210368$ different transitions.[25]

Feature #13: Multi-move (multi-tape)

Since we require that a multi-tape machine support no-move transitions for the various tapes, implementing a multi-move transition is extremely straightforward: on each tape, move a single cell in the desired direction (or stay in place for no-move), and then use a helper state to perform the remaining movements. Each tape's desired movements are processed independently, and once the head of the tape has performed its movements it performs no-move transitions while waiting for the other tape heads. When all tape heads have moved to their desired final positions, the machine switches to the actual target state of the transition.

Since how the heads of the different tapes move independently can be a bit tricky to understand in the abstract, let use examine a concrete example. Assume that a machine has three tapes (number `"0"`, `"1"`, and `"2"`) and a transition wants to move five cells to the left on the first tape, three cells to the right on the second tape, and one cell to the right on the third tape, while changing the state from $q_{\texttt{preparation}}$ to $q_{\texttt{success}}$. If the underlying machine does not support multi-move transitions, these would be handled as follows:

1. Transition from $q_{\texttt{preparation}}$ to $q_{\texttt{ω\_4L,2R,N\_ω\_success}}$ and move the head of the first tape left by one cell and the heads of the second and third tapes right one cell.

[25] As it happens, this exceeds the supported maximum size of a `Map` in JavaScript (jmrk 2019), and thus my implementation actually does not support it.

2. Transition from $q_{\omega\_4L,2R,N\_\omega\_success}$ to $q_{\omega\_3L,1R,N\_\omega\_success}$ and move the head of the first tape left by one cell, the head of the second tape right by one cell, and do not move the head of the third tape.

3. Transition from $q_{\omega\_3L,1R,N\_\omega\_success}$ to $q_{\omega\_2L,N,N\_\omega\_success}$ and move the head of the first tape left by one cell, the head of the second tape right by one cell, and do not move the head of the third tape.

4. Transition from $q_{\omega\_2L,N,N\_\omega\_success}$ to $q_{\omega\_1L,N,N\_\omega\_success}$ and move the head of the first tape left by one cell, moving neither the head of the second tape nor the head of the third tape.

5. Transition from $q_{\omega\_1L,N,N\_\omega\_success}$ to $q_{success}$ and move the head of the first tape left by one cell, moving neither the head of the second tape nor the head of the third tape.

As you can see, a series of helper states are used to combine multiple single-cell movements. The number of helper states needed is based on the magnitude of the largest movement of a tape head; the other tape heads reach their desired tape cell and then just wait there. Note that all of these intermediate transitions target all combinations of symbols (i.e. `"WILDCARD"`) and make no changes to the contents of the tapes.

Here, the type of the actual transition function changes in addition to the update to the type of the tuple the developer uses to specify transitions. The transition function type is:

$$Q \times ((\Gamma|\Gamma[]|\texttt{WILDCARD})^n|\texttt{WILDCARD}) \longrightarrow Q \times (\Gamma)^n \times (\{L, R, N, L(v), R(v)\})^n$$

and the tuple type is

$$Q \times ((\Gamma|\Gamma[]|\texttt{WILDCARD})^n|\texttt{WILDCARD}) \times Q \times (\Gamma|\texttt{NO\_CHANGE})^n \times (\{L, R, N, L(v), R(v)\})^n$$

Feature #14: Simulated extra tapes (multi-tape)

It is up to the developer to decide, when writing a program, how many tapes they want to make use of. In the example above, only two tapes were used, and since every machine that supports multiple tapes must support two tapes (the lowest number of tapes that is still "multiple") we could be sure that if the machine claimed it supported multiple tapes, it would have at least two. But, what happens if the developer wants to use more tapes than the machine supports? If the machine cannot just add extra tapes, we would be forced to simulate at least some of the tapes that the developer wants to use. It would be fairly confusing to add some extra complicated logic to use multiple tapes, where one or more of the tapes are used to simulate extra tapes, and other real tapes are used as individual tapes.[26] But, we have already identified

[26]See how confusing even the idea is?

the logic needed to simulate all of the tapes on one real tape. Accordingly, let us specify that, if a machine supports multiple real tapes but the developer wants to use more tapes, the compiler will simply simulate all of the tapes on a single real tape, leaving the rest of the real tapes unused.

Now that the developer is free to assume that multiple tapes are supported, we can develop a Brainfuck interpreter using multiple tapes, rather than needing to manually figure out how such an interpreter could work on a single tape.

4 Brainfuck

Brainfuck is a programming language invented by Urban Müller in 1993. The initial inspiration for the language came from discovering the "FALSE" programming language and the fact that the original compiler for the language was only 1024 bytes, which he considered "ridiculously small" (Müller 2017). Müller wanted to create a smaller compiler, and instead of optimizing a compiler for the "FALSE" language, opted to change the underlying problem and develop a new language (Müller 2017). The resulting compiler only required 240 bytes, which was later further reduced to under 200 bytes (Müller 2017). Müller named the new language "Brainfuck".

Brainfuck is a Turing-complete language with a very limited instruction set.[27] Because of the limitations imposed by the instruction set (no variables other than the single data array of integers, no functions at all, and no conditionals other than a `WHILE` loop), it can be categorized as a "Turing-tarpit", meaning a language that is Turing-complete but hard to use and write programs in (attempting to write such programs would "fuck your brain") (Chandra 2014, 119).

Brainfuck has no formal specification. As Müller has explained, he "never submitted it to the ANSI committee, which means there were lots and lots of variants." (Müller 2017) Instead, it was simply released with an initial working version. Accordingly, I need to identify which specific variant of Brainfuck I am trying to implement - the version that I will be working with is described in the following pages.

4 (a) Language

There are eight commands that are used to operate on an infinitely long tape of cells[28] extending to the right/in the positive direction. These cells all start with the value `"0"`, and can hold values between `"0"` and `"255"` (inclusive). Cells can be incremented or decremented; incrementing `"255"` overflows to `"0"`, and decrementing `"0"` underflows to `"255"`. The program has a data pointer, indicating where on the tape the program is currently operating, which begins

[27] For details of the Turing-completeness of Brainfuck, see, e.g. (Faase n.d.-b) for a high level explanation, though it depends on a few undocumented assumptions about the flavor of Brainfuck used. Cristofani explains those assumptions and confirms that the version of Brainfuck with limited cell size but an unbounded array of cells is indeed Turing-complete (Cristofani n.d.). This version of Brainfuck is also an extension of the P'' language which has been more formally proven to be Turing-complete (Böhm 1964); see the appendix on Brainfuck and P'' for more details. Further discussion of the different flavors of Brainfuck and their Turing-completeness can be found on the *esoteric programming languages wiki* page about the Brainfuck language: (Esolang contributors 2024).

[28] Though the original implementation only provided 30,000 cells, its documentation noted that the language was only Turing-complete "if we ignore the [tape] size limit" (Müller 1993, "readme"). Accordingly, I implement it with an infinitely-long tape to permit Turing-completeness.

at the first cell (index 0) and can move right or left. When trying to move left, if the data pointer is already at the first cell the data pointer remains in place on the first cell.

Separate from the data pointer, there is an instruction pointer identifying the command to execute. Other than the two jumping commands, after the execution of a command the instruction pointer moves to the following command. Thus, the set of instructions and the set of data values can be thought of as two separate tapes that the program moves across; it is a good thing we just figured out how to simulate extra tapes in our Turing Machine.

The eight commands in Brainfuck are all single characters and are executed as follows:

Command	Meaning
`>`	Move the data pointer to the right
`<`	Move the data pointer to the left
`+`	Increment the cell currently pointed at by the data pointer
`-`	Decrement the cell currently pointed at by the data pointer
`[`	If the current cell is `"0"`, move the instruction pointer forward past the corresponding `"]"`
`]`	Move the instruction pointer back to the corresponding `"["`
`.`	Output the character with the ASCII code from the current cell
`,`	Read a character in and store its ASCII code in the current cell

When reading input, EOF (end of file) is indicated by writing `"-1"` (which underflows to `"255"`).[29] There are other variations of Brainfuck that are equally Turing-complete, that may have unbounded cell sizes, a limited number of cells, different behavior on EOF, or other cases of divergent behavior (Cristofani n.d.). I had to choose a specific version to implement, so I went with the version where data values were bound with overflow and underflow, EOF results in `"-1"`, and there is an infinite number of cells. Formal operational semantics for this version of Brainfuck have been included in an appendix.

Before diving into the actual implementation, I should clarify what my goal is. I want to generate a single set of states and transitions that will execute arbitrary Brainfuck code given as input to the Turing Machine, not to develop a tool to translate Brainfuck code into states and transitions. Such a tool would

[29]To be clear, there is thus no difference in behavior between the program attempting to read in a character when there are none remaining, and the program reading in a character with code `"255"`. The original ASCII standard only used seven bits, i.e. the values `"0"` through `"127"` (Cerf 1969). As a result, writing `"255"` does not restrict the input values that are *documented* as supported (i.e. ASCII codes) (Müller 1993, "readme"). While this implementation does support reading in values that would not be valid ASCII codes (because they require eight bits, i.e. values `"128"` through `"255"`) developers who rely on such support should keep in mind that if the result of a read operation is that the data value holds `"255"`, either a character was actually read in, or the end of the input was reached.

necessarily produce different results for different Brainfuck programs, essentially compiling the Brainfuck into a Turing Machine specification. If the tool was written in a higher-level language, the Brainfuck program would no longer be the input to the machine. On the other hand, since a Turing Machine can compute anything programming languages can, it would be possible to have the tool itself running on the Turing Machine. It would then generate states and transitions, which would be read by another part of the tool to actual execute the Brainfuck. However, compiling the Brainfuck before executing it seems unnecessarily complicated. Accordingly, the states and transitions I develop are intended to interpret Brainfuck as input and run it, not convert it to a Turing Machine specification.

4 (b) Implementation: Overview

Now, how to implement Brainfuck? It is going to be (a bit) trickier than the other example programs described above. The various terms (tape names, parts of the input, etc.) that I define in this overview are summarized at the end of the section for reference.

Let's start with the simple commands, moving the data pointer along the data tape and manipulating data values. We need to process the instructions "`>`" and "`<`" for movement, and the instructions "`+`" and "`-`" for manipulating the values. Thankfully, now that we can assume our Turing Machine supports execution on multiple tapes, we can keep the program instructions and the data values entirely separate. Let the first tape (tape `0`, the "program tape") be used for the instructions, since that tape will start with the input to the Turing Machine that will contain the program itself.

Let us create a second tape (tape `1`, the "data tape") to be used to hold the infinitely long series of data values. If the size of data values is fixed, we do not need a specific marker for where each value starts. But, to make it easier to implement versions of Brainfuck that use different sized values (a potential future improvement), everything other than the automatic creation of new values (which has to know how big the values are) will try to infer the value-size from the contents of the tape. Accordingly, let's specify that values are separated by a helper symbol, "`;`", except that after the last initialized value, a "`/`" indicates that subsequent values still need to be initialized. Such an indicator is needed because while the Brainfuck code starts with data values containing "`0`", Turing Machines start with cells being entirely empty. To indicate that the Brainfuck program's data pointer is currently at a specific data value, the head of the data tape will point to the "`;`" *directly before* (to the left of) the tape cells holding the value.

We also need to decide how to encode the values of the Brainfuck data. Using the base-10 decimal system, while perhaps simpler for someone trying to examine the machine, leads to difficulties implementing overflow when incrementing "`255`" or underflow when decrementing "`0`". We could also create a base-256 system where each possible value had its own dedicated symbol, but

then we would need 256 different transitions defined for incrementing a cell, and another 256 for decrementing! Since the total range of the cell, 256, is a power of two, a system that is base-2 (or whose base is a power of two) would not suffer the base-10 problem of overflow and underflow, and would certainly be easier to implement. In this case, I chose to use hexadecimal (base-16), where *each Brainfuck data value* would be made of *multiple Turing Machine tape data cells*.[30] For values in the range 0-255, two data tape cells would be needed, but as long as we don't hard-code the size anywhere other than the initialization of new values, it should be pretty easy to change.

I also include a "$" at the start of the data tape, to make the absolute start of the tape clearer to someone looking at the tape. For example, if so far the program has put the value 72 (0x48) in the first location in memory and the value 69 (0x45) in the second, with the program currently looking at the third value, the data tape might look like the following:

$	;	4	8	;	4	5	;	0	0	/	␣

Now, how to add support for looping with "[" and "]"? Because loops can be nested, finding the "matching" opening/closing brace when jumping over cells is not as simple as just finding the next instance of the corresponding symbol. And, because the loops can be nested arbitrarily deep, we cannot store the depth/number of corresponding symbols to skip past in the state name. However, our example of determining whether there are more "0"s or "1"s in a string already showed how this can be solved: we can add an additional tape to be used to keep track of the number of currently open loops, moving right and left on it as needed.

Specifically, this third tape (tape 2, the "jump tape") will start with a marker cell to indicate the left bound. Let's re-use "$" for that. When not currently performing loop logic, the tape head for this tape will just sit and point at the "$". However, when we get to a "[" (where the current data value is "0") and we need to perform a jump, we will move right on this third tape, to an empty cell. We can then scan right along the program tape - each time we encounter *another* "[", which indicates another loop to jump over, the tape head for this jump tape will move one cell to the right, i.e. increasing the count of open loops. On the other hand, when we encounter a "]", the head of the jump tape will move one cell to the left. If we land on the "$", then clearly the matching "]" was found, and the jump is done; if not, we continue, having decreased the count of open loops by one.

We handle the "]" command similarly, except that we do not need to check if the current data value is "0", since the jump is unconditional.[31] When jumping back, each "]" increases the number of open loops, and each "["

[30]For clarity, I adopt the convention of prefixing hexadecimal values with "0x" to differentiate them from decimal values. Thus, "0xFF" is equivalent to the value "255".

[31]The "]" command can be treated as either a *unconditional jump* back to the corresponding "[", or as a *conditional jump* that is only executed when the current data value is not "0". I implement it as an unconditional jump for the sake of simplicity.

decreases the count, until we land on the corresponding "[" where the jump ends, as indicated by the head of the jump tape reaching the "$".[32]

To be clear, we explicitly assume that the program given is well-formed, meaning that every "[" has a corresponding "]". There must not be any extra "[" commands, nor any extra "]" commands.

Finally, we need to determine how to handle output (via ".") and input (via ","). We could add some complicated logic to convert input symbols to their hexadecimal ASCII values, for example, but going the other way would cause problems: not all ASCII values can be printed.[33] Plus, if we later switch to larger data values, all of the input and output handling will need to be rewritten. Instead, let's just read in and write out the raw hexadecimal values of the cells when requested, and leave it to the machine operator to try and understand them.[34]

But how do we actually read those values? Turing Machines have no extra input mechanism for a second type of input, where values will be read in by the program during execution. They only accept a single input, the contents of the tape at the start of execution (Sipser 2013, 166, 185).

Thankfully, such a scenario has already been faced by others writing programs that accept Brainfuck code as input but then somehow need to also read user input without a second input mechanism (such as Brainfuck self-interpreters). Officially, meaning based on Urban Müller's original implementation, any symbol in Brainfuck code other than one of the known commands above is ignored and can be used for comments (Müller 1993, "readme"). As a result, some implementations have taken the approach, which I will adopt, of declaring that any symbols in the Brainfuck code after the first "!" symbol should be considered the user input to the Brainfuck program itself rather than to the interpreter running the program (Cristofani n.d.). While this user input is not interactive, it is still input to the program.

However, we need to be able to read the user input symbols without losing track of where in the program we currently are. If we leave the user input symbols at the end of the program, when we try to read them we would need to jump to those symbols, and then we would have a hard time trying to find our way back to where we were. We could add some extra logic to handle this.[35] But why complicate things? We can just declare a fourth tape (tape

[32]Yes, if the data value is "0" a jump back will be immediately followed by a jump forward, but it is easier to implement an unconditional jump back than a conditional jump back, and we do not care about performance.

[33]For example, ASCII code "0x07" indicates that a bell should be rung (Cerf 1969, 4). However, we have not shown how to simulate a bell on our machine.

[34]Such an approach gives rise to a situation that does not occur in implementations written in higher-level programming languages (or even in Brainfuck): what if some, but not all, of the hexadecimal characters needed to fill a single data value are provided? In that case, such an input is invalid and implementations are free to do whatever they want; mine treats all missing hexadecimal characters as "F" for the sake of simplicity.

[35]For example, when we encounter a ",", replace it with some new symbol, perform the input, and then return to the only instance of the new symbol in the program, switching it

3, the "user input tape"), and move all of the inputs to that tape at the start of execution. Then, to read a value, we just need to advance the head of the user input tape to the next input value and copy that over to the data tape. The head of the first tape (where the program code is) doesn't need to jump around at all!

The same logic for conversion to and from ASCII applies to the output command. Accordingly, let us just specify that the raw hexadecimal values will be output. However, just like Turing Machines have no extra input mechanism, they have no output mechanism. The only output is the state that the machine halts in (Sipser 2013, 166). Instead, let us define a fifth tape (tape 4, the "output tape"), and say that the "output" of the machine is anything that gets written to that tape.

To summarize, we will have five tapes. The first will hold the program code, with the head of the tape identifying the instruction to execute. The second will hold the data values, separated by ";"s except that the last one is followed by a "/", and the head of the tape will point to the ";" to the left of the current data value. The third tape will be used to implement jumps, and will normally just contain a "$"; during jumps the location of the tape head will reflect the number of loops that are currently open. The fourth tape will hold the hexadecimal input to the Brainfuck program, which will be considered anything after the first "!" in the input to the Turing Machine. Finally, the fifth tape will be considered the output tape, and the hexadecimal characters written there will be the output of the Brainfuck program.

To avoid confusion and to summarize the descriptions above, in discussing the implementation of Brainfuck the following terms have the given meanings:

- **Program tape**: the first tape (index #0) that will hold the Brainfuck program; until the setup is complete, this tape will also hold any user input, which will come after the program with a "!" separating the two
- **Data tape**: the second tape (index #1) that will hold the Brainfuck program's data
- **Jump tape**: the third tape (index #2) that will be used to track jumps
- **User input tape**: the fourth tape (index #3) that will hold user input until it is read
- **Output tape**: the fifth tape (index #4) that will hold the output of the program
- **Command**: one of the symbols +, -, <, >, ., ,, [, or] (note that though the different options listed here are separated by commas, the comma symbol is itself a valid command)
- **Program**: the series of commands that make up the Brainfuck program

back to "," before continuing.

- **Data cell**: a single Turing Machine cell on the data tape that holds a hexadecimal value; part of a data value
- **Data value**: an individual value (from the perspective of the Brainfuck code) made up of a series of data cells
- **User input**: the input *to the Brainfuck program* that may be specified at the end of the input *to the TM* by appending a "!" and then a series of hexadecimal values to the end of the Brainfuck program
- **Data pointer**: pointer that identifies which data value is used by commands; between commands, indicated by marking the ";" to the left of the data value as the head of the data tape
- **Instruction pointer**: pointer that identifies which command is being executed; indicated by the head of the program tape
- **Initialized data value**: A data value that the TM has initialized such that each data cell holds "0"; once initialized a data value's cells can be changed, but must remain hexadecimal values

The indices of the various tapes are vital because they identify the order in which the symbols to target and the tape updates to make must appear in a transition. In the simple case of transitions on two tapes where symbols and movements are written as <a, b> it should already be clear that this is not equivalent to <b, a >. The order of the elements of the sets of symbols or movements for a transition must be the order of the tapes that the symbol or movement is meant to apply to.

4 (c) Implementation: Setup

Of course, the entire purpose of this paper is to explain how to translate higher-level algorithms like the one above into individual states and transitions that a Turing Machine will be able to process. In an appendix, I include the generation of the actual states and transitions (making use of all of the various features introduced previously) for executing Brainfuck in the manner that I explained above, but it is also useful to go into a bit more depth on how the implementation actually works.

The states are essentially split into two groups: setting up the tapes based on the user-provided program and input, and then actually executing that code. Before diving into those states, let us consider a small example program to understand how the language works and what will be needed to interpret it.

Let us consider a simple program that will

1. Print out an exclamation mark (ASCII code 33 = 0x21)
2. Read in a user input

3. Print that input
4. Print out another exclamation mark

where the user input is the pound character ("#", ASCII code 35 = 0x23).

One such implementation of this program would be:

Brainfuck example program

```
1  ++++
2  [
3            >
4            ++++ ++++
5            <
6            -
7  ]
8  >+
9  .
10 >,.
11 <.
12 !23
```

This program will:

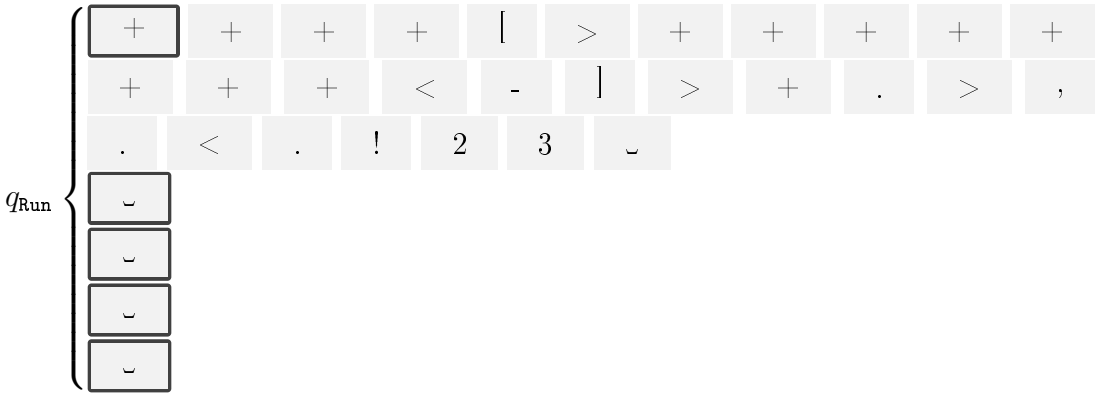
- [Line 1] Increment cell #0 four times (from "0" to "4").
- [Lines 2-7] Run a loop while the value in cell #0 is not zero, which will:
  - [Lines 3,4] Go right to cell #1 and increment in eight times.
  - [Lines 5,6] Go left back to cell #0 and decrement it once.
- [Line 8] After the loop, go right to cell #1 and increment it.
- [Line 9] Print the value in cell #1.
- [Line 10] Go right to cell #2, read input into the cell, and print it.
- [Line 11] Go left to cell #1 and print it again.

And after the program is the user input.

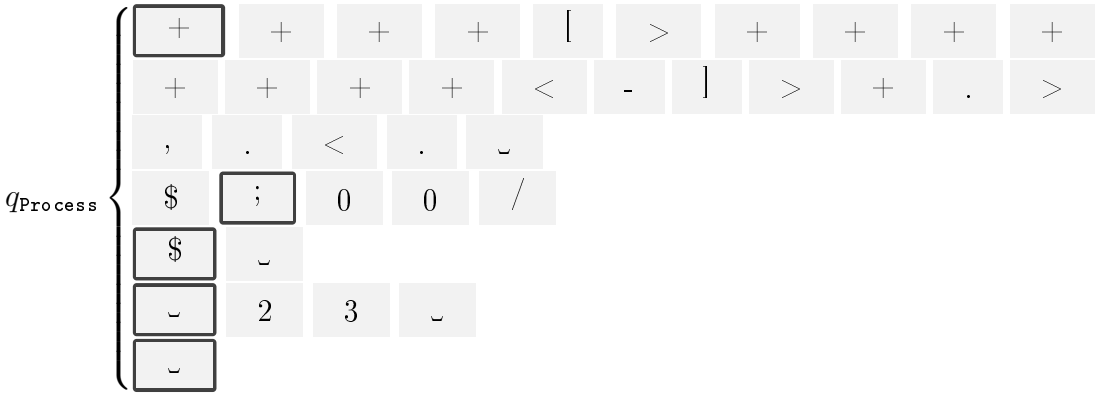
While the program above is spread out to make it more readable, the actual input to the tape has none of this spacing. Instead, the tape simply starts with the content "++++[>++++++++<-]>+.>,.<.!23".[36]

Thus, our five-tape machine will start with:

[36]For many implementations of Brainfuck including the spacing would have been fine, since non-command symbols are ignored (Müller 1993, "readme"). In my implementation I do not support comments, to make it simpler.



and we want to set it up so that when we get to the process state, the state is



The first step is moving the user input from the end of the program tape to the dedicated user input tape. To do so, we need to move the head of the program tape all the way to the right until we find any user input. But how to get back to the start? We have not added the gadgets to add a marker at the start of a tape to the multi-tape version of the machine.[37] While we could do so, the approach I went with is simpler to implement, if less elegant: since the third tape (the jump tape) will start with a "$", write that "$" at the start, and then during set up move the heads of the program tape and the jump tape together. Thus, the head of the program tape is at the start if and only if the head of the jump tape also gets back to the start.[38] Accordingly, our setup procedure begins with:

[37] As explained in footnote n.24 on page 55, adding gadgets would not be too hard, and indeed a prior version of the code had support for gadgets on multiple tapes. But, it turns out there is a simpler option here that does not require implementing gadgets, so I didn't bother proving how they could be simulated.

[38] This adds a new invariant to all of the transitions relating to setting up the Brainfuck handling: the head of the jump tape and the head of the program tape must always be in the same index cell, meaning that they are an equal number of cells to the right of the leftmost cell of the tape. This cannot be detected during the actual execution of the transitions, and is enforced by designing the transitions so that the tape heads have identical movements on the program tape and the jump tape.

1. Write a "$" at the head of the jump tape, moving none of the heads.

2. As long as the head of the program tape is over a valid command, move the heads of both the program tape and the jump tape to the right.

Once the program tape has reached the end of the program, it will encounter either a "!" (indicating that there is indeed user input, though it might be empty) or an empty cell (indicating no user input). If there is a "!", we will erase it and move on to copying the input; if not, we will move back to the start, the same way that we do after copying the user input.

Since we want the user input to be copied starting in the second cell of the user input tape (so that we can have the head of the user input tape point to an empty cell when not in the process of actually reading inputs, to simplify our transitions), we will also need to move the head of the user input tape over exactly once; we can do this in the first transition. Accordingly, our setup procedure is modified to be:[39]

1. Write a "$" at the head of the jump tape, moving the head of the user input tape one cell to the right and leaving the rest of the heads in place.

2. As long as the head of the program tape is over a valid command, move the heads of both the program tape and the jump tape to the right.

3. If the head of the program tape reaches an "!":

   (a) Replace the "!" with an empty cell, moving the heads of the program tape and the jump tape to the right.

   (b) As long as the head of the program tape is over a hex symbol (i.e. not an empty cell):

   - Copy that value to the cell at the head of the user input tape.
   - Erase that value from the cell on the program tape.
   - Move the heads of the program, jump, and user input tapes each one cell to the right.

   (c) After finding the first empty cell (marking the end of the user input), move the head of the user input tape back to the start (i.e. move left as long as it is over a non-empty cell).

4. With the head of the program tape at an empty cell (either because of no user input or because there was user input that got moved to the dedicated tape), the head of the jump tape at the same index, and the head of the user input tape at an empty cell (either at the start if user input was copied, or at the second cell if there was no user input), move the heads of both the program tape and the jump tape to the left until the jump tape head is on a "$", indicating that both heads are at the start of their tapes.

[39] As I build up the setup procedure, here too new parts and changes are underlined.

As a reminder, the head of the program tape is known to be at the same index as the head of the jump tape because we require that they move together so that we can keep track of when the program tape head returns to the start of the tape (after the setup is done).

We also need to initialize the data tape. By doing this initialization when we know the head of the program tape will be over an empty cell (i.e. at the end of processing user input and before returning back to the start of the tape) we can always do the data tape initialization with a consistent status of the other tapes.[40]

The first part of the setup, adding the "$", can be done at the same time that we add the "$" to the jump tape (since that transition is already set up to handle any of the commands on the program tape), and then we will actually set up the first data value after processing user input. Accordingly, let us modify the setup procedure to be:

1. Write a "$" at the heads of the data tape and jump tape, moving the head of the user input tape one to the cell right and leaving the rest of the heads in place.

2. As long as the head of the program tape is over a valid command, move the heads of both the program tape and the jump tape to the right.

3. If the head of the program tape reaches an "!":

   (a) Replace the "!" with an empty cell, moving the heads of the program tape and the jump tape to the right.

   (b) As long as the head of the program tape is over a hex symbol (i.e. not an empty cell):

   - Copy that value to the cell at the head of the user input tape.
   - Erase that value from the cell on the program tape.
   - Move the heads of the program, jump, and user input tapes each one cell to the right.

   (c) After finding the first empty cell (marking the end of the user input), move the head of the user input tape back to the start (i.e. move left as long as it is over a non-empty cell).

4. With the head of the program tape at an empty cell (either because of no user input or because there was user input that got moved to the dedicated tape), the head of the jump tape at the same index, and the head of the user input tape at an empty cell (either at the start if user

[40] Yes, we can do initialization with the head of the program tape at the start, but that requires transitions to match against all of the possible Brainfuck instructions. While we don't care about performance, there is no reason to require storing eight times as many transitions when we can do it with one, by doing the setup of the data tape while the head of the program tape is known to be pointing at an empty cell.

input was copied, or at the second cell if there was no user input), set up the data tape:

   (a) Write a `";"` on the data tape and move over `C + 1` cells to the right, where `C` is the number of data cells to store a single Brainfuck data value.
   (b) Write a `"/"` on the data tape and move left.
   (c) While the data tape head is over an empty cell, write a `"0"` and move left.
   (d) Once the data tape head is back over the `";"`, leave it there.

5. With the head of the program tape at an empty cell (either because of no user input or because there was user input that got moved to the dedicated tape), the head of the data tape at the `";"`, the head of the jump tape at the same index, and the head of the user input tape at an empty cell (either at the start if user input was copied, or at the second cell if there was no user input), move the heads of both the program tape and the jump tape to the left until the jump tape head is on a `"$"`, indicating that both heads are at the start of their tapes

And this is everything that is needed for the setup! Once the heads of the program and jump tapes are back at the start, the situation for each tape will be (where "index" means the index of the cell where the tape head is located, not the index of the tape itself in the overall machine):

- Program tape: head is at first instruction (index 0).
- Data tape: head is at the `";"` (index 1) before the first and only initialized data value.
- Jump tape: head is at the `"$"` (index 0) indicating no jump is in progress.
- User input tape: head is at an empty cell before user input to be read; either index 0 (if there was input) or index 1 (if there wasn't) but it doesn't matter.
- Output tape: head is at the start (index 0).

4 (d) Implementation: Processing

Now we just need to do the processing. While each of the eight Brainfuck commands will obviously be processed differently, there are some common parts that shared states can be used for:

- Incrementing and decrementing both start at the rightmost data cell within the data value to update the last symbol, and then move left; in

the case of overflow (increment) or underflow (decrement) of the specific hexadecimal symbol, additional cells get updated, moving left through the data value. After the increment or decrement has finished, the tape needs to move left to return to the ";" before the data value.

- Checking if a data value is "0" (for conditional jumps) can be done moving across the data value in either direction; I implemented it moving left starting from the rightmost data cell of the data value.

- Moving to the right begins with finding the end of the current data value, and either initializes the next data value or simply returns to processing if the next value is already set up.

- Reading input and writing output both move right through the data value, handling data cells one at a time. When completed, the data tape head needs to move back to the start of that data value.

The primary logic for processing Brainfuck is in a state named "process". Let us build up the logic for it, one command at a time, starting with the "+" command to increment the data value:

- When the program command is to increment the data value:

    1. Move to the rightmost data cell for the data value.
    2. Increment the symbol in that data cell according to hexadecimal math rules and move left. If the increment was from "F" to "0", stay in a state indicating that incrementing should be performed, otherwise change states to indicate that the operation was completed.
    3. Once the operation is completed, so long as the head of the data tape is not at the ";" at the left side of the data value, move the tape head left. After reaching the ";", move the head of the program tape to the right; the command has been completed.

A lot of this logic is reused when we are decrementing:

- When the program command is to increment or decrement the data value:

    1. Move to the rightmost data cell for the data value.
    2. Increment or decrement, based on the current symbol on the program tape the symbol in that data cell according to hexadecimal math rules and move left. If the increment was from "F" to "0" or the decrement was from "0" to "F", stay in a state indicating that incrementing or decrementing should be performed, otherwise change states to indicate that the operation was completed.

3. Once the operation is completed, so long as the head of the data tape is not at the ";" at the left side of the data value, move the tape head left. After reaching the ";", move the head of the program tape to the right; the command has been completed.

On the other hand, the procedure to read input or print output moves across the data value in the opposite direction, i.e. from the leftmost data cell to the rightmost. Starting with reading input:

- When the program command is to read input into the data value:

  1. Move right once on the data tape to the leftmost data cell for the data value, and move right once on the user input tape.
  2. As long as the head of the data tape is not over a ";" or "/" indicating the end of the data value, replace the existing symbol in the data cell with the symbol from the user input tape (or "F" if there is no remaining user input), erase the symbol from the user input tape, and move the heads of both the data tape and the user input tape to the right.
  3. After reaching the ending ";" or "/", move the head of the data tape back to the starting ";" and then move the head of the program tape to the right; the command has been completed.

When we add output:

- When the program command is to read input into the data value or to print the current contents of the data value:

  1. Move right once on the data tape to the leftmost data cell for the data value, and on input only, move right once on the user input tape.
  2. As long as the head of the data tape is not over a ";" or "/" indicating the end of the data value:
     - When reading input, replace the existing symbol in the data cell with the symbol from the user input tape (or "F" if there is no remaining user input), erase the symbol from the user input tape, and move the heads of both the data tape and the user input tape to the right.
     - When performing output, copy the symbol in the data cell to the output tape, and move the heads of both the data tape and the output tape to the right.
  3. After reaching the ending ";" or "/", move the head of the data tape back to the starting ";" and then move the head of the program tape to the right; the command has been completed.

Next, let us implement moving between data values. Moving left is fairly trivial, since we do not need to initialize new data values. The only potential complication is when the data pointer is on the very first data value, in which case the data pointer should stay in the same place. As I noted previously, a "$" exists at the start of the data tape; we can check if we landed on that after the transition to identify if the head of the data tape was already at the ";" before the first data value.

- When the program command is to move to the data value to the left:
    1. Move the head of the data tape one cell to the left.
    2. If the head of the data tape is on the "$" (meaning that we were just at the first data value) move the data tape head back to the right (i.e. onto the ";")
    3. Otherwise keep moving the data tape head left until the previous ";".
    4. Move the head of the program tape to the right; the command has been completed.

On the other hand, moving right can be complicated. If the next data value was already initialized, we simply need to find the next ";". But, if the next data value is not yet initialized, this is when the initialization needs to happen. We already saw how to do that for the first data value when setting up the data tape. Thus, moving right can be implemented as:

- When the program command is to move to the data value to the right:
    1. Move the head of the data tape one cell to the right.
    2. While the head of the data tape has not reached the end of the data value, keep moving right.
    3. Upon reaching a ";", stay there and move the head of the program tape to the right; we finished moving to the next data value.
    4. Otherwise, upon reaching a "/", initialize the data value:
        (a) Replace the "/" on the data tape with ";" and move over `C + 1` cells to the right, where `C` is the number of data cells to store a single Brainfuck data value.
        (b) Write a "/" on the data tape and move left.
        (c) While the data tape is over an empty cell, write a "0" and move left.
        (d) Once the data tape is back over the ";", leave it there and move the head of the program tape to the right; we finished moving to the next data value.

Finally, let us implement jumping. As a reminder, the input is explicitly assumed to be valid, i.e. all "["s and "]"s are paired up and there are no unmatched jumps. Since the "]" is being implemented as an unconditional jump back to the matching "[", we want to end up **on** that "[" so that it gets processed to implement the condition. As discussed above, we will use the jump tape to keep track of the number of currently open loops. Accordingly:

- When the program command is to jump back:
    1. Move the head of the program tape to the left, and the head of the jump tape (which was at the starting "$") to the right.
    2. For any program command other than a "[" or "]", move the program tape head left and leave the jump tape head in place.
    3. For any "]", move the program tape head left and the jump tape head right, indicating another open loop.
    4. For any "[", move the program tape head left and the jump tape head left, indicating one fewer open loops.
    5. Once the jump tape head reaches the "$" at the start (i.e. right after the transition on the matching "["), move the program tape head to the right (onto that "[") and resume normal processing.

Jumping forward, however, is a bit trickier - we only want to jump forward if the current data value is zero. We also want to jump to the instruction *after* the corresponding "]", since if we jump to that "]" we would just jump back and get caught in an infinite loop. Accordingly:

- When the program command is to jump forward:
    1. Move the head of the data tape to the rightmost data cell for the data value.
    2. As long as the data cell holds a "0", move left on the data tape and stay in the same processing state.
    3. If the data cell holds another hexadecimal value, the data value is not zero; return to the ";" at the start of the data value and then move the program tape head to the right and proceed with normal processing; no jump is needed.
    4. Otherwise, if the data tape head reaches the ";", the data value is "0", and a jump is needed:
        (a) Move the head of the program tape to the right, and the head of the jump tape (which was at the starting "$") to the right.
        (b) For any program command other than a "[" or "]", move the program tape head right and leave the jump tape head in place.

(c) For any "]", move the program tape head right and the jump tape head left, indicating one fewer open loops.

(d) For any "[", move the program tape head right and the jump tape head right, indicating another open loop.

(e) Once the jump tape head reaches the "$" at the start (i.e. right after the transition on the matching "]"), resume normal processing, leaving the program tape head on the command after the "]".

And finally, perhaps the easiest transition for the program tape:

- When the program tape cell is empty, indicating the end of the program, halt and accept.

And that is it! Putting it all together, we now have the logic to actually execute the Brainfuck. The full details of the different operations are simplified below to avoid simply repeating everything covered above; after each operation (other than "[" or "]") move the head of the program tape to the right to proceed to the next instruction.

- When the program command is to increment or decrement the data value:

  1. Move to the rightmost data cell for the data value.
  2. Increment or decrement, based on the current symbol on the program tape the symbol in that data cell according to hexadecimal values and move left. If the increment was from "F" to "0" or the decrement was from "0" to "F", stay in a state indicating that incrementing or decrementing should be performed, otherwise change states to indicate that the operation was completed.
  3. Once the operation is completed, move the data tape head left to the ";".

- When the program command is to read input or print a data value:

  1. Move right once to the leftmost data cell for the data value, and on input only, move right once on the user input tape.
  2. As long as the head of the data tape is not over a ";" or "/" indicating the end of the data value:
     - When reading input, replace the existing symbol in the data cell with the symbol from the user input tape (or "F" if there is no remaining user input), erase the symbol from the user input tape, and move the heads of both the data tape and the user input tape to the right.

   - When performing output, copy the symbol in the data cell to the output tape, and move the heads of both the data tape and the output tape to the right.
3. After reaching the ending `";"` or `"/"`, move the head of the data tape back to the starting `";"`.

- When the program command is to move to the data value to the left:
  1. Move the head of the data tape one cell to the left.
  2. If the head of the data tape is on the `"$"` move the data tape head back to the right.
  3. Otherwise keep moving the data tape head left until the previous `";"`.

- When the program command is to move to the data value to the right:
  1. Move the head of the data tape one cell to the right.
  2. While the head of the data tape has not reached the end of the data value, keep moving right.
  3. Upon reaching a `";"`, stay there.
  4. Otherwise, upon reaching a `"/"`, initialize the data value:
     (a) Replace the `"/"` on the data tape with `";"` and move over `C + 1` cells to the right, where `C` is the number of data cells to store a single Brainfuck data value.
     (b) Write a `"/"` on the data tape and move left.
     (c) While the data tape is over an empty cell, write a `"0"` and move left.
     (d) Once the data tape is back over the `";"`, stay there.

- When the program command is to jump forward:
  1. Move the head of the data tape to the rightmost data cell for the data value.
  2. As long as the data cell holds a `"0"`, move left on the data tape.
  3. If the data cell holds another hexadecimal value, return to the `";"` at the start of the data value and stay there.
  4. Otherwise, if the data tape head reaches the `";"`, the data value is 0, and a jump is needed:
     (a) Move the head of the program tape to the right, and the head of the jump tape (which was at the starting `"$"`) to the right.
     (b) For any program command other than a `"["` or `"]"`, move the program tape head right and leave the jump tape head in place.

(c) For any "]", move the program tape head right and the jump tape head left.

(d) For any "[", move the program tape head right and the jump tape head right.

(e) Once the jump tape head reaches the "$" at the start, resume normal processing.

- When the program command is to jump back:

1. Move the head of the program tape to the left, and the head of the jump tape (which was at the starting "$") to the right.
2. For any program command other than a "[" or "]", move the program tape head left and leave the jump tape head in place.
3. For any "]", move the program tape head left and the jump tape head right.
4. For any "[", move the program tape head left and the jump tape head left.
5. Once the jump tape head reaches the "$" at the start, move the program tape head to the right and resume normal processing.

- When the program tape cell is empty, halt and accept.

As long as you consider the previously-described handling of input and output to the Brainfuck code to be valid given the constraints a Turing Machine imposes, that is everything you need to have a Turing Machine treat its input like a Brainfuck program and execute it! The actual states and transitions needed are generated in an appendix.

While in the title of this thesis I claimed to describe simulating Brainfuck on a *single-taped* Turing Machine, the explanation I just gave and the appendix with the actual implementation rely on using five different tapes. Did I lie in the title? I do not believe so - I have previously explained how to have a single-taped Turing Machine simulate all of the functionality of a multi-taped Turing Machine. Thus, even if I did not explicitly list the convoluted transitions for the Brainfuck interpreter to be run on a single-taped machine, I have shown *how* to determine those transitions. The code that I have attached to this thesis with my implementation of the compiler is fully capable of generating these individual transitions, given enough time and memory.

5 What next

Having demonstrated *how* a Turing Machine can be set up to execute Brainfuck code, the question then becomes, what next? Brainfuck is an esoteric programming language, and being able to execute it does not demonstrate how to execute any other language, like JavaScript, Python, or C++. What is the point of implementing Brainfuck?

The answer to that is twofold. First, there are existing tools that can generate Brainfuck, either from other languages or from a set of macros, to make it easier to program in. Second, in the process of implementing Brainfuck I clearly explained *how* a developer can be given extra shortcuts for the functionality of a Turing Machine without actually changing the machine itself, which will make it easier to demonstrate implementing other languages.

5 (a) Generating Brainfuck

Brainfuck is a hard language to program in. I have been using the language for years and even I need to manually trace the execution of all but the simplest programs. However, there exist tools that make it easier to do so, either by providing features that translate into Brainfuck (the same way that the features I described above for developers configuring Turing Machines can be translated into individual states and transitions for the machine to process) or by converting some other language into Brainfuck.

Frans Faase's introduction to Brainfuck (available online at `https://www.iwriteiam.nl/Ha_bf_intro.html`) defines a series of macros to add perform common operations, such as copy a number from one cell to another, multiply two cells together, and add if-then-else conditionals, among other features (Faase n.d.-a). Tufts' own professor Alva Couch has created a Perl script to implement those macros, as well as to permit developers to define their own constants and macros (Couch n.d.). Other macro-based generators for Brainfuck are also available online.

Outside of tools to write Brainfuck code more easily, there also exist tools to convert code in other languages to Brainfuck. Tools exist for conversion from assembly ((Ackermann 2019; Gaponov 2021; Szewczyk, Maya, and Emilia 2022), and more), Python ((cmspeedrunner 2023)), and there is even a single tool that supports conversion from Java, C, and assembly: (Zattera 2021).

Even though some of these only support subsets of the other languages, they still make it much easier to write Brainfuck code. Instead of writing their Brainfuck instructions manually, developers can make use of a more familiar language like Java, or a series of macros.

5 (b) Summary

In the process of implementing Brainfuck, I created a much more usable interface for the developer to configure a Turing Machine. Rather than set up

individual transitions that can only move left or right by a single cell and only target a single symbol, the developer now has the ability to program a multi-tape machine, where transitions can be defined that target multiple possible symbols (or even a wildcard, to indicate that a transition should target all symbol combinations), and the head of a tape can move multiple cells in either direction as well as a single cell, or even stay in the same place. While many of these features have likely been discussed elsewhere previously, my hope is that I have clearly proven *how* these features provide no additional *computational* functionality. I demonstrated exactly how to set up a single-taped Turing machine to simulate each of these features; they merely provide shortcuts for developers.

5 (c) Future work

In the process of expanding the interface that the developer has access to, there were a few additional places were adding shortcuts might have been helpful, and may be useful in the future. While the potential for gadgets on a multi-tape machine has been mentioned previously, another feature that I did not note before is allowing a per-symbol callback to be used when defining transitions. For example, when the compiler is simulating multiple tapes, there are a number of states where whatever symbol is found simply gets marked as the head of the simulated tape. Instead of listing out the transition for each of these symbols, I could imagine a developer who provided a callback function that was specific to the state, and just accepted an input symbol and returned a symbol. The type of such a tuple would be:

$$Q \times (\Gamma|\Gamma[]|\texttt{WILDCARD}) \times Q \times (\Gamma \longrightarrow \Gamma) \times \{L, R, N, L(v), R(v)\})$$

Such a feature would be useful when defining a number of similar transitions for the same state, where the transitions all end in the same state and with the same movement and the only question is what symbol to write to the tape. Other places this could be useful include implementing incrementing/decrementing operations or converting letters to upper or lower case. While such a feature was not needed for implementing Brainfuck (though technically none of the features are "needed") it may be useful for those interested in implementing some other higher-level language.

If a developer wants to implement a higher-level language, like Forth or Lisp (both of which I considered implementing before deciding on Brainfuck), it should now be easier to do so while still being satisfied that the implementation would work on a single-tape Turing Machine without any of the extra features I present. All of the preparation work I discussed above to add the extra features does not need to be repeated in future works regarding implementing other languages. The developer is free to assume that there are an arbitrary number of tapes and take advantage of the more usable transition definitions, while still being secure in the knowledge that the interface developed provides no extra computational functionality.

References

All URLs should be available via the Wayback Machine. All GitHub sources include the commit id of the version used.

Ackermann, Hilmar. "HELVM/Brainfuckasmcompiler." GitHub, April 22, 2019. https://github.com/helvm/BrainfuckAsmCompiler. Commit e290878.

Böhm, Corrado. "On a Family of Turing Machines and the Related Programming Language." *ICC Bulletin* 3, no. 3 (July 1964): 185–94.

Carlini, Nicholas, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. "Control-Flow Bending: On the Effectiveness of Control-Flow Integrity." In *Proceedings of the 24th USENIX Security Symposium*, 2015. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/carlini.

Cerf, Virt. "ASCII Format for Network Interchange." RFC editor, October 16, 1969. https://www.rfc-editor.org/info/rfc0020. STD 80. RFC 20. https://doi.org/10.17487/RFC0020.

Chandra, Vikram. *Geek Sublime: The beauty of code, the code of beauty*. Minneapolis, MN: Gray Wolf Press, 2014. https://archive.org/details/geeksublimebeaut0000chan.

Churchill, Alex, Stella Biderman, and Austin Herrick. "*Magic: The Gathering* Is Turing Complete." *Computing Research Repository*, April 23, 2019. https://doi.org/10.48550/arXiv.1904.09828.

Cmspeedrunner. "Cmspeedrunner/Pyf." GitHub, March 31, 2023. https://github.com/cmspeedrunner/Pyf. Commit 7fc1bea.

Cook, Matthew. "Universality in Elementary Cellular Automata." *Complex Systems* 15, no. 1 (2004): 1–40. https://content.wolfram.com/sites/13/2023/02/15-1-1.pdf.

Couch, Alva. Bfmacro: A bf macro-interpreter. Accessed May 1, 2024. https://www.cs.tufts.edu/~couch/bfmacro/bfmacro/.

Cristofani, Daniel B. Yet another brainfuck reference. Accessed May 1, 2024. https://brainfuck.org/brainfuck.html.

Esolang contributors. "Brainfuck." Esolang, April 30, 2024. https://esolangs.org/wiki/Brainfuck. Revision 127058.

Faase, Frans. An introduction to programming in BF. Accessed May 1, 2024. https://www.iwriteiam.nl/Ha_bf_intro.html.

———. BF is Turing-complete. Accessed May 1, 2024. https://www.iwriteiam.nl/Ha_bf_Turing.html.

Gaponov, Yaroslav. "Yaroslavgaponov/As2bf." GitHub, February 12, 2021. https://github.com/YaroslavGaponov/as2bf. Commit e9dbd35.

jmrk. "Maximum Number of Entries in Node.Js Map? [Answer]." Stack Overflow, January 31, 2019. https://stackoverflow.com/a/54466812.

Müller, Urban. "dev/lang/brainfuck-2.lha" Aminet, June 9, 1993. https://aminet.net/package/dev/lang/brainfuck-2.

———. "Brainfuck or How I Learned to Change the Problem." YouTube, June 13, 2017. https://www.youtube.com/watch?v=gjm9irBs96U.

Ostrovsky, Igor. "Human Heart Is a Turing Machine, Research on Xbox 360 Shows. Wait, What?" Igor Ostrovsky Blogging, September 24, 2009. https://igoro.com/archive/human-heart-is-a-turing-machine-research-on-xbox-360-shows-wait-what/.

Poss, Raphael. "On the Turing-Completeness of C *(cont.)*." dr knz @ work, October 20, 2012. https://dr-knz.net/on-the-turing-completeness-of-c-part-2.html.

Rendell, Paul. "Turing Machine Universality of the Game of Life." *SpringerLink*. Thesis, Springer Cham, 2016. https://link.springer.com/book/10.1007/978-3-319-19842-2.

Scarle, Simon. "Implications of the Turing Completeness of Reaction-Diffusion Models, Informed by GPGPU Simulations on an Xbox 360: Cardiac Arrhythmias, Re-Entry and the Halting Problem." *Computational Biology and Chemistry* 33, no. 4 (August 2009): 253–60. https://doi.org/10.1016/j.compbiolchem.2009.05.001.

Sipser, Michael. *Introduction to the Theory of Computation*. 3rd ed. Boston, MA: Cengage Learning, 2013.

Szewczyk, Kamila, Maya, and Eliza Emilia. "Kspalaiologos/Asmbf." GitHub, October 23, 2022. https://github.com/kspalaiologos/asmbf. Commit 3545adb.

Turing, Alan. "On Computable Numbers, with an Application to the Entscheidungsproblem." *Proceedings of the London Mathematical Society*, 2, 42, no. 1 (January 1937): 230–65. https://doi.org/10.1112/plms/s2-42.1.230.

Zattera, Massimiliano. "Mzattera/Fuckbench." GitHub, January 19, 2021. https://github.com/mzattera/FuckBench. Commit f7040ad.

A Appendix: Constants and symbols

The following constants are available for use in the pseudocode. When the value is given as `IMP`, this means that there is no specific underlying value but rather is used to indicate processing that the compiler or machine should perform.

Name	Value	Notes
`EMPTY_CELL`		An empty cell, for defining transitions
`STATE_ACCEPT`	ACCEPT	The accept state name
`STATE_REJECT`	REJECT	The reject state name
`L`	`IMP`	Movement to the left
`R`	`IMP`	Movement to the right
On its own, `L` represents moving a single cell to the left, but `L(n)` can be used to indicate moving `n` cells; the same works for movement to the right with `R`.		
`NO_CHANGE`	`IMP`	A transition should not change the tape symbol
`WILDCARD`	`IMP`	A transition targets all tape alphabet symbols
`N`	`IMP`	No movement of the tape head

Additionally, the following symbols are used to simulate multiple tapes. The names listed here are not used in the pseudocode examples, but only used within the attached code implementation. Instead, the values shown below are used - they are used consistently throughout this document with the given meanings, but since the details of how multiple tapes are simulated are hidden from the developer, implementations need not represent them the same way.

Name	Display	Significance
`MULTITAPE_REAL_START`	`{^^}`	Marker for the start of the real tape that is holding multiple simulated tapes
`MULTITAPE_BOUND_L`	`{<}`	Marker for the left boundary of a simulated tape
`MULTITAPE_BOUND_R`	`{>}`	Marker for the right boundary of a simulated tape
`MULTITAPE_AFTER_LAST`	`{$$}`	Marker after the end of the last simulated tape

B Appendix: States created by the compiler

Below are the different forms of state names that the compiler will generate for various features. Since all of the state names will include a "ω" to prevent conflicts between compiler-generated state names and developer-provided state names and symbol names, we only need to prevent conflicts between different compiler-generated state names. There are places where I may have gone overboard in including multiple "ω"s but there is no harm in guaranteeing that there are no conflicts.

As a reminder, the use of "ω" here is merely for documentation purposes, and it may be any symbol that is not found in a user-provided state name. Additionally, when a symbol is converted to a string for state names, the compiler will add a "S:" prefix to any developer-provided symbol that is at risk of clashing with a compiler-used symbol such as "{e}" for the empty cell.

For a single tape:

- For no-move and multi-move transitions:
  `ω_{movement remaining}_ω_{target state}_ω`
- For the cell-shifting gadget:
  `ω_paste{symbol}_ω_{movement}_ω_{target state}_ω`
- For inserting a first cell marker:
  `ω_firstCellMarker_ω_{start state}_ω_{match content}_ω_findEnd`
  `ω_firstCellMarker_ω_{start state}_ω_{match content}_ω_doShifts`
- For inserting symbols after a specific cell marker:
  `ω_afterCellMarker_ω_{target state}_ω_{match content}_findEnd`
  `ω_afterCellMarker_ω_{target state}_ω_{match content}_doShifts`
  `ω_afterCellMarker_ω_{target state}_ω_{match content}_doInsert`

For setting up the simulation of multiple tapes and providing the infrastructure needed for multiple tapes regardless of the specific states and symbols that the developer is using:[41]

- "`ω_init_{state}`", where "`stage`" is one of:
  - "`insertLeftBound`" - inserting a "{<}" at the left end of the first simulated tape.
  - "`insertEmpty`" - inserting a "␣" that will hold the name of the simulated state during post-transition updates.
  - "`findEnd`" - finding the tape contents, which will become the end of the first simulated tape.

[41] Note that many of these states are implemented using gadgets that generate further state names; only the top-level state names are listed since any states generated by a simulated gadget are the responsibility of that gadget.

  - "afterEnd" - used to jump $3n + 1$ cells to the right of the end of the first simulated tape, where n is the number of extra tapes to add (i.e. one less than the number of desired simulated tapes).[42]
  - "afterLast" - writing a "{$$}" after the last simulated tape.
  - "atBoundR" - writing a "{>}" at the end of a simulated tape.
  - "atExtraHead" - writing a "$\underline{\sqcup}$" at the head of a simulated tape.
  - "atBoundL" - writing a "{<}" at the start of a simulated tape.
  - "findStart" - returning to the "{^^}" after adding the additional simulated tapes.
  - "atStart" - finding the start of the first simulated tape, which needs to be marked as the head of the tape.
  - "markFirstHead" - marking the first cell of the first simulated tape as the head of that tape.

- "ω_doEmptyCellInsertions" and "ω_doEmptyCellInsertions-{step}", for inserting extra empty cells at the right end of simulated tapes if needed, where "step" is one of:
  - "search" - finding any "{$\bar{>}$}", or the "{$$}" after the last simulated tape.
  - "done" - returning to the start of the real tape or performing another empty cell insertion if needed.
  - "findToGrab" - finding a empty cell to shift over from after the last simulated tape.
  - "foundToGrab" - inserting a "{>}" after the "{$\bar{>}$}".
  - "justInserted" - jumping back to that "{$\bar{>}$}".
  - "nowMakeEmptyHead" - replacing the "{$\bar{>}$}" with "$\underline{\sqcup}$".
- "ω_process", for switching into the individual simulation of a specific state.

For simulating multiple tapes:

- When at the start of simulating a specific state: "ωs_{original state}ω".
- When matching against successive simulated tape heads: "ωs_{original state}ω_f_ω{symbol}ω".

[42] Each extra simulated tape needs three cells to begin with, to hold the left bound of the tape ("{<}"), the empty cell at the head of the tape ("$\underline{\sqcup}$"), and the right bound of the tape ("{<}"). Finally, after the last tape a cell is needed for the marker that indicates the end of the simulated tapes ("{$$}"). Thus, a total of three cells per extra tape, and an additional one cell regardless of the number of extra tapes, will be set up.

  - After the second matched symbol, the state is changed to one that ends with "`_f_ω{symbol1}ω_ω{symbol2}ω`", each additional matched symbol gets added to the end of the state name.

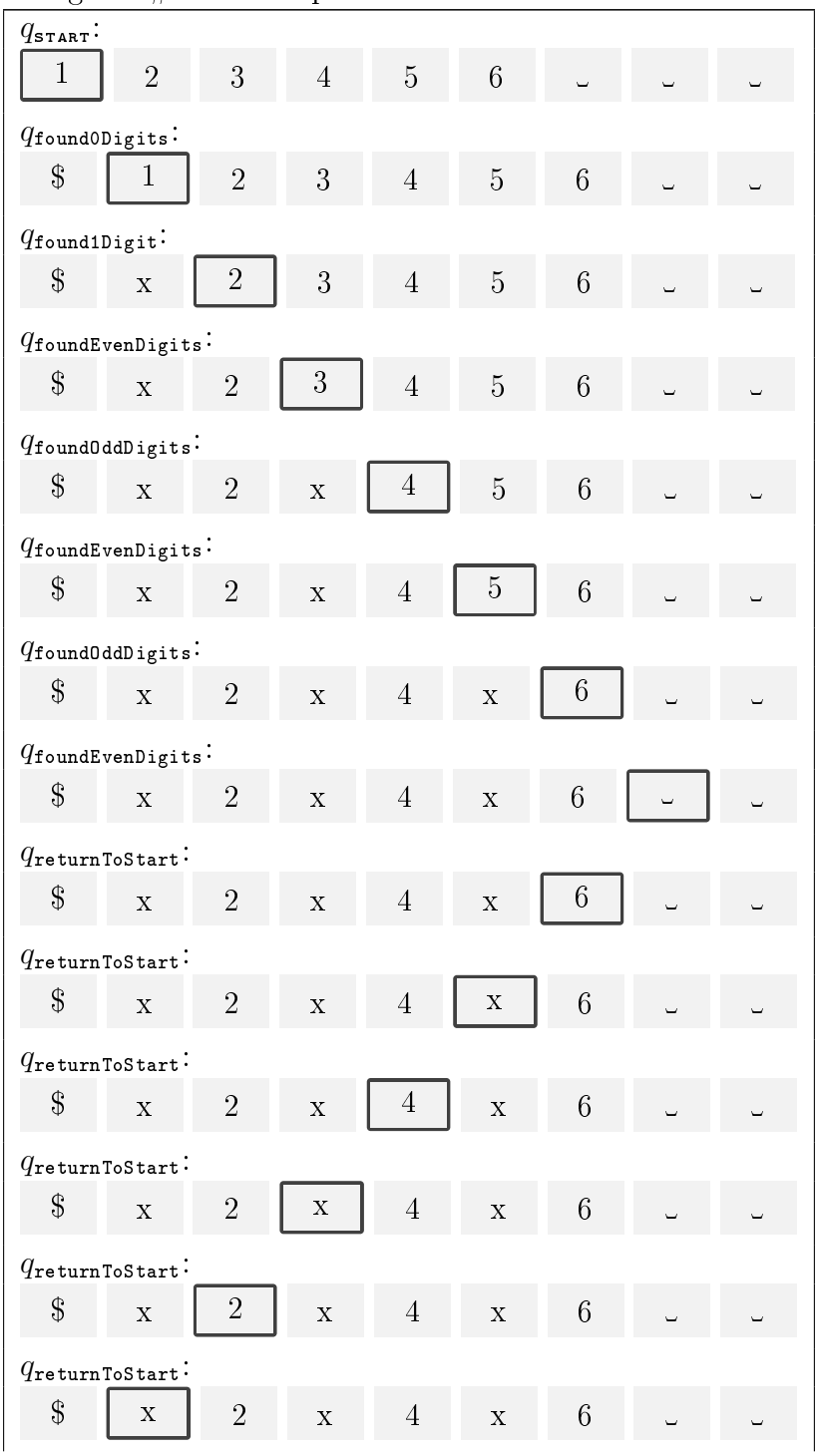
- After applying the updates to simulated additional tapes: "`ωa_{new state}ω`".
  - Before the updates have all being applied, they are stored in the state name: "`ωa_{new state}ω_p_ω{s1},{m1}ω_ω{s2},{m2}ω`", where "`s2`" is the new symbol to write on the second tape and "`m2`" is the movement to apply to that tape; after each tape update is applied it is removed from the end of the state name.
  - After each update is applied, before the update is removed from the state name, the machine first switches to a state with a name in the format "`ωa_{new state}ω_p_ω{s1},{m1}ω_ω{s2},{m2}ω-markHead`" to mark the new simulated head of the tape.
  - Before the updates have all being applied, they are stored in the state name: "`ωa_{new state}ω_p_ω{s1},{m1}ω_ω{s2},{m2}ω`", where "`s2`" is the new symbol to write on the second tape and "`m2`" is the movement to apply to that tape; after each tape update is applied it is removed from the end of the state name.

- When simulating multi-move transitions: "`ω_{moves}_ω_{new state}_ω`", where "`moves`" encodes the set of remaining moves to apply to the simulated tapes.

C Appendix: Power-of-two checker execution

For the program to check if a series of digits has a power-of-two length, the execution of the input "123456" would look as follows:

Program #19 with input "123456":



$q_{\texttt{returnToStart}}$:

$	x	2	x	4	x	6	␣	␣

$q_{\texttt{found0Digits}}$:

$	x	2	x	4	x	6	␣	␣

$q_{\texttt{found0Digits}}$:

$	x	2	x	4	x	6	␣	␣

$q_{\texttt{found1Digit}}$:

$	x	x	x	4	x	6	␣	␣

$q_{\texttt{found1Digit}}$:

$	x	x	x	4	x	6	␣	␣

$q_{\texttt{foundEvenDigits}}$:

$	x	x	x	4	x	6	␣	␣

$q_{\texttt{foundEvenDigits}}$:

$	x	x	x	4	x	6	␣	␣

$q_{\texttt{foundOddDigits}}$:

$	x	x	x	4	x	x	␣	␣

$q_{\textsc{reject}}$:

$	x	x	x	4	x	x	␣	␣

On the other hand, if there is a power-of-two number of digits, as with "7890", then the machine would accept:

Program #19 with input "7890":

$q_{\textsc{start}}$:

1	2	3	4	5	6	␣	␣	␣

$q_{\texttt{found0Digits}}$:

$	1	2	3	4	5	6	␣	␣

$q_{\texttt{found1Digit}}$:

$	x	2	3	4	5	6	␣	␣

$q_{\texttt{foundEvenDigits}}$:

$	x	2	3	4	5	6	␣	␣

$q_{\texttt{foundOddDigits}}$:

$	x	2	x	[4]	5	6	␣	␣

$q_{\texttt{foundEvenDigits}}$:

$	x	2	x	4	[5]	6	␣	␣

$q_{\texttt{foundOddDigits}}$:

$	x	2	x	4	x	[6]	␣	␣

$q_{\texttt{foundEvenDigits}}$:

$	x	2	x	4	x	6	[␣]	␣

$q_{\texttt{returnToStart}}$:

$	x	2	x	4	x	[6]	␣	␣

$q_{\texttt{returnToStart}}$:

$	x	2	x	4	[x]	6	␣	␣

$q_{\texttt{returnToStart}}$:

$	x	2	x	[4]	x	6	␣	␣

$q_{\texttt{returnToStart}}$:

$	x	2	[x]	4	x	6	␣	␣

$q_{\texttt{returnToStart}}$:

$	x	[2]	x	4	x	6	␣	␣

$q_{\texttt{returnToStart}}$:

$	[x]	2	x	4	x	6	␣	␣

$q_{\texttt{returnToStart}}$:

[$]	x	2	x	4	x	6	␣	␣

$q_{\texttt{found0Digits}}$:

$	[x]	2	x	4	x	6	␣	␣

$q_{\texttt{found0Digits}}$:

$	x	[2]	x	4	x	6	␣	␣

$q_{\texttt{found1Digit}}$:

$	x	x	[x]	4	x	6	␣	␣

$q_{\texttt{found1Digit}}$:

$	x	x	x	[4]	x	6	␣	␣

$q_{\texttt{foundEvenDigits}}$:

$	x	x	x	4	[x]	6	␣	␣

$q_{\texttt{foundEvenDigits}}$:

$	x	x	x	4	x	[6]	␣	␣

$q_{\texttt{foundOddDigits}}$:

\$	x	x	x	4	x	x	␣	␣

$q_{\textsc{reject}}$:

\$	x	x	x	4	x	x	␣	␣

D Appendix: Multi-tape simulation program

The following is the expanded version of the program to compare the number of "0"s and "1"s in an input string, making use of two tapes. The additional tape is being simulated by the compiler; the output below is for after the simulation of the additional tape but before expanding the various gadgets and other features. If the output made use of no extra features, it would be significantly longer.

Program #23: Require equal "0"s and "1"s (*compiler view* - two tapes)

Given a series of "0"s and "1"s, accept if there is an equal number of each digit, reject otherwise.

In state $q_{\texttt{START}}$:

- On any symbol, insert a "{^^}", shifting all input symbols to the right, then move right to the second cell, ending in state $q_{\texttt{ω\_init\_insertLeftBound}}$

In state $q_{\texttt{ω\_init\_insertLeftBound}}$:

- On any symbol, insert a "{<}" after a "{^^}", then move right to the next cell, ending in state $q_{\texttt{ω\_init\_insertEmpty}}$

In state $q_{\texttt{ω\_init\_insertEmpty}}$:

- On any symbol, insert a "␣" after a "{^^}", then move right to the next cell, ending in state $q_{\texttt{ω\_init\_findEnd}}$

In state $q_{\texttt{ω\_init\_findEnd}}$:

- On symbol "␣", replace with "␣", move right, and change to state $q_{\texttt{ω\_init\_afterEnd}}$
- On any other symbol, leave symbol unchanged, move right, and change to state $q_{\texttt{ω\_init\_findEnd}}$

In state $q_{\texttt{ω\_init\_afterEnd}}$:

- On symbol "␣", replace with "{>}", move right four cells, and change to state $q_{\texttt{ω\_init\_afterLast}}$

In state $q_{\texttt{ω\_init\_afterLast}}$:

- On symbol "␣", replace with "{$$}", move left, and change to state $q_{\texttt{ω\_init\_atBoundR}}$

In state $q_{\omega\_init\_atBoundR}$:
- On symbol "␣", replace with "{>}", move left, and change to state $q_{\omega\_init\_atExtraHead}$
- On symbol "{>}", replace with "{>}", move left, and change to state $q_{\omega\_init\_findStart}$

In state $q_{\omega\_init\_atExtraHead}$:
- On symbol "␣", replace with "$\bar{␣}$", move left, and change to state $q_{\omega\_init\_atBoundL}$

In state $q_{\omega\_init\_atBoundL}$:
- On symbol "␣", replace with "{<}", move left, and change to state $q_{\omega\_init\_atBoundR}$

In state $q_{\omega\_init\_findStart}$:
- On symbol "{^^}", replace with "{^^}", move right, and change to state $q_{\omega\_init\_atStart}$
- On any other symbol, leave symbol unchanged, move left, and change to state $q_{\omega\_init\_findStart}$

In state $q_{\omega\_init\_atStart}$:
- On symbol "␣", replace with "Q:{Run}", move right two cells, and change to state $q_{\omega\_init\_markFirstHead}$

In state $q_{\omega\_init\_markFirstHead}$:
- On symbol "0", replace with "$\bar{0}$", move left two cells, and change to state $q_{\omega\_process}$
- On symbol "1", replace with "$\bar{1}$", move left two cells, and change to state $q_{\omega\_process}$
- On symbol "$", replace with "$\bar{\$}$", move left two cells, and change to state $q_{\omega\_process}$
- On symbol "␣", replace with "$\bar{␣}$", move left two cells, and change to state $q_{\omega\_process}$

In state $q_{\omega\_process}$:
- On symbol "Q:{Run}", replace with "␣", move right, and change to state $q_{\omega s\_qRun\omega}$
- On symbol "Q:{more0s}", replace with "␣", move right, and change to state $q_{\omega s\_qmore0s\omega}$
- On symbol "Q:{more1s}", replace with "␣", move right, and change to state $q_{\omega s\_qmore1s\omega}$

In state $q_{\omega\_doEmptyCellInsertions}$:

- On symbol "{<}", replace with "{<}", move right, and change to state $q_{\omega\_doEmptyCellInsertions\text{-}search}$

In state $q_{\omega\_doEmptyCellInsertions\text{-}search}$:

- On symbol "{$$}", replace with "{$$}", move left, and change to state $q_{\omega\_doEmptyCellInsertions\text{-}done}$
- On symbol "{$\bar{>}$}", replace with "{$\bar{>}$}", move right, and change to state $q_{\omega\_doEmptyCellInsertions\text{-}findToGrab}$
- On any of the symbols ["0", "1", "$", "␣", "$\bar{0}$", "$\bar{1}$", "$\bar{\$}$", "$\bar{␣}$", "{<}", "{>}"], leave symbol unchanged, move right, and change to state $q_{\omega\_doEmptyCellInsertions\text{-}search}$

In state $q_{\omega\_doEmptyCellInsertions\text{-}findToGrab}$:

- On symbol "{$$}", replace with "{$$}", do not move the tape head, and change to state $q_{\omega\_doEmptyCellInsertions\text{-}foundToGrab}$
- On any of the symbols ["0", "1", "$", "␣", "$\bar{0}$", "$\bar{1}$", "$\bar{\$}$", "$\bar{␣}$", "{<}", "{>}", "{$\bar{>}$}"], leave symbol unchanged, move right, and change to state $q_{\omega\_doEmptyCellInsertions\text{-}findToGrab}$

In state $q_{\omega\_doEmptyCellInsertions\text{-}foundToGrab}$:

- On any symbol, insert a "{>}" after a "{$\bar{>}$}", then move right to the next cell, ending in state $q_{\omega\_doEmptyCellInsertions\text{-}justInserted}$

In state $q_{\omega\_doEmptyCellInsertions\text{-}justInserted}$:

- On any of the symbols ["{<}", "{$$}"], leave symbol unchanged, move left two cells, and change to state $q_{\omega\_doEmptyCellInsertions\text{-}nowMakeEmptyHead}$

In state $q_{\omega\_doEmptyCellInsertions\text{-}nowMakeEmptyHead}$:

- On symbol "{$\bar{>}$}", replace with "$\bar{␣}$", move left, and change to state $q_{\omega\_doEmptyCellInsertions\text{-}done}$

In state $q_{\omega\_doEmptyCellInsertions\text{-}done}$:

- On symbol "{$\bar{>}$}", replace with "{$\bar{>}$}", move right, and change to state $q_{\omega\_doEmptyCellInsertions\text{-}findToGrab}$
- On symbol "Q:{ACCEPT}", replace with "Q:{ACCEPT}", do not move the tape head, and change to state $q_{\texttt{ACCEPT}}$
- On symbol "Q:{REJECT}", replace with "Q:{REJECT}", do not move the tape head, and change to state $q_{\texttt{REJECT}}$
- On symbol "Q:{Run}", replace with "Q:{Run}", do not move the tape head, and change to state $q_{\omega\_process}$
- On symbol "Q:{more0s}", replace with "Q:{more0s}", do not move the tape head, and change to state $q_{\omega\_process}$
- On symbol "Q:{more1s}", replace with "Q:{more1s}", do not move the tape head, and change to state $q_{\omega\_process}$
- On any of the symbols ["0", "1", "$", "␣", "$\bar{0}$", "$\bar{1}$", "$\bar{\$}$", "$\bar{␣}$", "{<}", "{>}"], leave symbol unchanged, move left, and change to state $q_{\omega\_doEmptyCellInsertions\text{-}done}$

In state $q_{\omega s\_qRun\omega}$:

- On symbol "$\bar{␣}$", replace with "$\bar{␣}$", move right, and change to state $q_{\omega s\_qRun\omega\_f\_\omega\{e\}\omega}$
- On symbol "$\bar{0}$", replace with "$\bar{0}$", move right, and change to state $q_{\omega s\_qRun\omega\_f\_\omega 0\omega}$
- On symbol "$\bar{1}$", replace with "$\bar{1}$", move right, and change to state $q_{\omega s\_qRun\omega\_f\_\omega 1\omega}$
- On any of the symbols ["0", "1", "$", "␣", "{>}", "{<}"], leave symbol unchanged, move right, and change to state $q_{\omega s\_qRun\omega}$

In state $q_{\omega s\_qRun\omega\_f\_\omega\{e\}\omega}$:

- On symbol "$\bar{␣}$", replace with "$\bar{␣}$", move right, and change to state $q_{\omega a\_ACCEPT\omega\_p\_\omega\{e\},1R\omega\_\omega\$,1L\omega}$
- On any of the symbols ["0", "1", "$", "␣", "{>}", "{<}"], leave symbol unchanged, move right, and change to state $q_{\omega s\_qRun\omega\_f\_\omega\{e\}\omega}$

In state $q_{\omega a\_ACCEPT\omega\_p\_\omega\{e\},1R\omega}$:

- On any of the symbols ["0", "1", "$", "␣", "{<}", "{>}"], leave symbol unchanged, move left, and change to state $q_{\omega a\_ACCEPT\omega\_p\_\omega\{e\},1R\omega}$
- On any of the symbols ["$\bar{0}$", "$\bar{1}$", "$\bar{\$}$", "$\bar{␣}$"], replace with "␣", move right, and change to state $q_{\omega a\_ACCEPT\omega\_p\_\omega\{e\},1R\omega\text{-}markHead}$

In state $q_{\omega a\_ACCEPT\omega\_p\_\omega\{e\},1R\omega\text{-}markHead}$:

- On symbol "{<}", replace with "{<}", move right, and change to state $q_{\omega a\_ACCEPT\omega\_p\_\omega\{e\},1R\omega\text{-}markHead}$
- On symbol "{>}", replace with "{$\bar{>}$}", move left, and change to state $q_{\omega a\_ACCEPT\omega}$
- On symbol "0", replace with "$\bar{0}$", move left, and change to state $q_{\omega a\_ACCEPT\omega}$
- On symbol "1", replace with "$\bar{1}$", move left, and change to state $q_{\omega a\_ACCEPT\omega}$
- On symbol "$", replace with "$\bar{\$}$", move left, and change to state $q_{\omega a\_ACCEPT\omega}$
- On symbol "␣", replace with "$\bar{␣}$", move left, and change to state $q_{\omega a\_ACCEPT\omega}$

In state $q_{\omega a\_ACCEPT\omega\_p\_\omega\{e\},1R\omega\_\omega\$,1L\omega}$:

- On any of the symbols ["0", "1", "$", "␣", "{<}", "{>}"], leave symbol unchanged, move left, and change to state $q_{\omega a\_ACCEPT\omega\_p\_\omega\{e\},1R\omega\_\omega\$,1L\omega}$
- On any of the symbols ["$\bar{0}$", "$\bar{1}$", "$\bar{\$}$", "$\bar{␣}$"], replace with "$", move left, and change to state $q_{\omega a\_ACCEPT\omega\_p\_\omega\{e\},1R\omega\_\omega\$,1L\omega\text{-}markHead}$

In state $q_{\omega a\_ACCEPT\omega\_p\_\omega\{e\},1R\omega\_\omega\$,1L\omega\text{-}markHead}$:

- On symbol "{<}", replace with "{<}", move right, and change to state $q_{\omega a\_ACCEPT\omega\_p\_\omega\{e\},1R\omega\_\omega\$,1L\omega\text{-}markHead}$
- On symbol "{>}", replace with "{$\bar{>}$}", move left, and change to state $q_{\omega a\_ACCEPT\omega\_p\_\omega\{e\},1R\omega}$
- On symbol "0", replace with "$\bar{0}$", move left, and change to state $q_{\omega a\_ACCEPT\omega\_p\_\omega\{e\},1R\omega}$
- On symbol "1", replace with "$\bar{1}$", move left, and change to state $q_{\omega a\_ACCEPT\omega\_p\_\omega\{e\},1R\omega}$
- On symbol "$", replace with "$\bar{\$}$", move left, and change to state $q_{\omega a\_ACCEPT\omega\_p\_\omega\{e\},1R\omega}$
- On symbol "␣", replace with "$\bar{␣}$", move left, and change to state $q_{\omega a\_ACCEPT\omega\_p\_\omega\{e\},1R\omega}$

In state $q_{\omega a\_ACCEPT\omega}$:

- On symbol "{<}", replace with "{<}", move left, and change to state $q_{\omega a\_ACCEPT\omega\text{-}doRecord}$
- On any of the symbols ["0", "1", "$", "␣"], leave symbol unchanged, move left, and change to state $q_{\omega a\_ACCEPT\omega}$

In state $q_{\omega a\_ACCEPT\omega\text{-}doRecord}$:

- On symbol "␣", replace with "Q:{ACCEPT}", move right, and change to state $q_{\omega\_doEmptyCellInsertions}$

In state $q_{\omega s\_qRun\omega\_f\_\omega 0\omega}$:
- On symbol "$\bar{\sqcup}$", replace with "$\bar{\sqcup}$", move right, and change to state $q_{\omega a\_qmore0s\omega\_p\_\omega 0,1R\omega\_\omega\$,1R\omega}$
- On any of the symbols ["0", "1", "$", "␣", "{>}", "{<}"], leave symbol unchanged, move right, and change to state $q_{\omega s\_qRun\omega\_f\_\omega 0\omega}$

In state $q_{\omega a\_qmore0s\omega\_p\_\omega 0,1R\omega}$:
- On any of the symbols ["0", "1", "$", "␣", "{<}", "{>}"], leave symbol unchanged, move left, and change to state $q_{\omega a\_qmore0s\omega\_p\_\omega 0,1R\omega}$
- On any of the symbols ["$\bar{0}$", "$\bar{1}$", "$\bar{\$}$", "$\bar{\text{␣}}$"], replace with "0", move right, and change to state $q_{\omega a\_qmore0s\omega\_p\_\omega 0,1R\omega\text{-}markHead}$

In state $q_{\omega a\_qmore0s\omega\_p\_\omega 0,1R\omega\text{-}markHead}$:
- On symbol "{<}", replace with "{<}", move right, and change to state $q_{\omega a\_qmore0s\omega\_p\_\omega 0,1R\omega\text{-}markHead}$
- On symbol "{>}", replace with "{$\bar{>}$}", move left, and change to state $q_{\omega a\_qmore0s\omega}$
- On symbol "0", replace with "$\bar{0}$", move left, and change to state $q_{\omega a\_qmore0s\omega}$
- On symbol "1", replace with "$\bar{1}$", move left, and change to state $q_{\omega a\_qmore0s\omega}$
- On symbol "$", replace with "$\bar{\$}$", move left, and change to state $q_{\omega a\_qmore0s\omega}$
- On symbol "␣", replace with "$\bar{\text{␣}}$", move left, and change to state $q_{\omega a\_qmore0s\omega}$

In state $q_{\omega a\_qmore0s\omega\_p\_\omega 0,1R\omega\_\omega\$,1R\omega}$:
- On any of the symbols ["0", "1", "$", "␣", "{<}", "{>}"], leave symbol unchanged, move left, and change to state $q_{\omega a\_qmore0s\omega\_p\_\omega 0,1R\omega\_\omega\$,1R\omega}$
- On any of the symbols ["$\bar{0}$", "$\bar{1}$", "$\bar{\$}$", "$\bar{\text{␣}}$"], replace with "$", move right, and change to state $q_{\omega a\_qmore0s\omega\_p\_\omega 0,1R\omega\_\omega\$,1R\omega\text{-}markHead}$

In state $q_{\omega a\_qmore0s\omega\_p\_\omega 0,1R\omega\_\omega\$,1R\omega\text{-}markHead}$:
- On symbol "{<}", replace with "{<}", move right, and change to state $q_{\omega a\_qmore0s\omega\_p\_\omega 0,1R\omega\_\omega\$,1R\omega\text{-}markHead}$
- On symbol "{>}", replace with "{$\bar{>}$}", move left, and change to state $q_{\omega a\_qmore0s\omega\_p\_\omega 0,1R\omega}$
- On symbol "0", replace with "$\bar{0}$", move left, and change to state $q_{\omega a\_qmore0s\omega\_p\_\omega 0,1R\omega}$
- On symbol "1", replace with "$\bar{1}$", move left, and change to state $q_{\omega a\_qmore0s\omega\_p\_\omega 0,1R\omega}$
- On symbol "$", replace with "$\bar{\$}$", move left, and change to state $q_{\omega a\_qmore0s\omega\_p\_\omega 0,1R\omega}$
- On symbol "␣", replace with "$\bar{\text{␣}}$", move left, and change to state $q_{\omega a\_qmore0s\omega\_p\_\omega 0,1R\omega}$

In state $q_{\omega a\_qmore0s\omega}$:
- On symbol "{<}", replace with "{<}", move left, and change to state $q_{\omega a\_qmore0s\omega\text{-}doRecord}$
- On any of the symbols ["0", "1", "$", "␣"], leave symbol unchanged, move left, and change to state $q_{\omega a\_qmore0s\omega}$

In state $q_{\omega a\_qmore0s\omega\text{-}doRecord}$:
- On symbol "␣", replace with "Q:{more0s}", move right, and change to state $q_{\omega\_doEmptyCellInsertions}$

In state $q_{\omega s\_qRun\omega\_f\_\omega 1\omega}$:
- On symbol "$\bar{␣}$", replace with "$\bar{␣}$", move right, and change to state $q_{\omega a\_qmore1s\omega\_p\_\omega 1,1R\omega\_\omega\$,1R\omega}$
- On any of the symbols ["0", "1", "$", "␣", "{>}", "{<}"], leave symbol unchanged, move right, and change to state $q_{\omega s\_qRun\omega\_f\_\omega 1\omega}$

In state $q_{\omega a\_qmore1s\omega\_p\_\omega 1,1R\omega}$:
- On any of the symbols ["0", "1", "$", "␣", "{<}", "{>}"], leave symbol unchanged, move left, and change to state $q_{\omega a\_qmore1s\omega\_p\_\omega 1,1R\omega}$
- On any of the symbols ["$\bar{0}$", "$\bar{1}$", "$\bar{\$}$", "$\bar{␣}$"], replace with "1", move right, and change to state $q_{\omega a\_qmore1s\omega\_p\_\omega 1,1R\omega\text{-}markHead}$

In state $q_{\omega a\_qmore1s\omega\_p\_\omega 1,1R\omega\text{-}markHead}$:
- On symbol "{<}", replace with "{<}", move right, and change to state $q_{\omega a\_qmore1s\omega\_p\_\omega 1,1R\omega\text{-}markHead}$
- On symbol "{>}", replace with "{$\bar{>}$}", move left, and change to state $q_{\omega a\_qmore1s\omega}$
- On symbol "0", replace with "$\bar{0}$", move left, and change to state $q_{\omega a\_qmore1s\omega}$
- On symbol "1", replace with "$\bar{1}$", move left, and change to state $q_{\omega a\_qmore1s\omega}$
- On symbol "$", replace with "$\bar{\$}$", move left, and change to state $q_{\omega a\_qmore1s\omega}$
- On symbol "␣", replace with "$\bar{␣}$", move left, and change to state $q_{\omega a\_qmore1s\omega}$

In state $q_{\omega a\_qmore1s\omega\_p\_\omega 1,1R\omega\_\omega\$,1R\omega}$:
- On any of the symbols ["0", "1", "$", "␣", "{<}", "{>}"], leave symbol unchanged, move left, and change to state $q_{\omega a\_qmore1s\omega\_p\_\omega 1,1R\omega\_\omega\$,1R\omega}$
- On any of the symbols ["$\bar{0}$", "$\bar{1}$", "$\bar{\$}$", "$\bar{␣}$"], replace with "$", move right, and change to state $q_{\omega a\_qmore1s\omega\_p\_\omega 1,1R\omega\_\omega\$,1R\omega\text{-}markHead}$

In state $q_{\omega a\_qmore1s\omega\_p\_\omega1,1R\omega\_\omega\$,1R\omega\text{-}markHead}$:

- On symbol "{<}", replace with "{<}", move right, and change to state $q_{\omega a\_qmore1s\omega\_p\_\omega1,1R\omega\_\omega\$,1R\omega\text{-}markHead}$
- On symbol "{>}", replace with "{$\bar{>}$}", move left, and change to state $q_{\omega a\_qmore1s\omega\_p\_\omega1,1R\omega}$
- On symbol "0", replace with "$\bar{0}$", move left, and change to state $q_{\omega a\_qmore1s\omega\_p\_\omega1,1R\omega}$
- On symbol "1", replace with "$\bar{1}$", move left, and change to state $q_{\omega a\_qmore1s\omega\_p\_\omega1,1R\omega}$
- On symbol "$", replace with "$\bar{\$}$", move left, and change to state $q_{\omega a\_qmore1s\omega\_p\_\omega1,1R\omega}$
- On symbol "␣", replace with "$\bar{␣}$", move left, and change to state $q_{\omega a\_qmore1s\omega\_p\_\omega1,1R\omega}$

In state $q_{\omega a\_qmore1s\omega}$:

- On symbol "{<}", replace with "{<}", move left, and change to state $q_{\omega a\_qmore1s\omega\text{-}doRecord}$
- On any of the symbols ["0", "1", "$", "␣"], leave symbol unchanged, move left, and change to state $q_{\omega a\_qmore1s\omega}$

In state $q_{\omega a\_qmore1s\omega\text{-}doRecord}$:

- On symbol "␣", replace with "Q:{more1s}", move right, and change to state $q_{\omega\_doEmptyCellInsertions}$

In state $q_{\omega s\_qmore0s\omega}$:

- On symbol "$\bar{␣}$", replace with "$\bar{␣}$", move right, and change to state $q_{\omega s\_qmore0s\omega\_f\_\omega\{e\}\omega}$
- On symbol "$\bar{0}$", replace with "$\bar{0}$", move right, and change to state $q_{\omega s\_qmore0s\omega\_f\_\omega0\omega}$
- On symbol "$\bar{1}$", replace with "$\bar{1}$", move right, and change to state $q_{\omega s\_qmore0s\omega\_f\_\omega1\omega}$
- On any of the symbols ["0", "1", "$", "␣", "{>}", "{<}"], leave symbol unchanged, move right, and change to state $q_{\omega s\_qmore0s\omega}$

In state $q_{\omega s\_qmore0s\omega\_f\_\omega\{e\}\omega}$:

- On symbol "$\bar{\$}$", replace with "$\bar{\$}$", move right, and change to state $q_{\omega a\_ACCEPT\omega\_p\_\omega\{e\},1L\omega\_\omega\$,1L\omega}$
- On symbol "$\bar{␣}$", replace with "$\bar{␣}$", move right, and change to state $q_{\omega a\_REJECT\omega\_p\_\omega\{e\},1R\omega\_\omega\{e\},N\omega}$
- On any of the symbols ["0", "1", "$", "␣", "{>}", "{<}"], leave symbol unchanged, move right, and change to state $q_{\omega s\_qmore0s\omega\_f\_\omega\{e\}\omega}$

In state $q_{\omega a\_ACCEPT\omega\_p\_\omega\{e\},1L\omega}$:

- On any of the symbols ["0", "1", "$", "␣", "{<}", "{>}"], leave symbol unchanged, move left, and change to state $q_{\omega a\_ACCEPT\omega\_p\_\omega\{e\},1L\omega}$
- On any of the symbols ["$\bar{0}$", "$\bar{1}$", "$\bar{\$}$", "$\bar{␣}$"], replace with "␣", move left, and change to state $q_{\omega a\_ACCEPT\omega\_p\_\omega\{e\},1L\omega\text{-}markHead}$

In state $q_{\omega a\_ACCEPT\omega\_p\_\omega\{e\},1L\omega\text{-}markHead}$:

- On symbol "{<}", replace with "{<}", move right, and change to state $q_{\omega a\_ACCEPT\omega\_p\_\omega\{e\},1L\omega\text{-}markHead}$
- On symbol "{>}", replace with "{$\bar{>}$}", move left, and change to state $q_{\omega a\_ACCEPT\omega}$
- On symbol "0", replace with "$\bar{0}$", move left, and change to state $q_{\omega a\_ACCEPT\omega}$
- On symbol "1", replace with "$\bar{1}$", move left, and change to state $q_{\omega a\_ACCEPT\omega}$
- On symbol "$", replace with "$\bar{\$}$", move left, and change to state $q_{\omega a\_ACCEPT\omega}$
- On symbol "␣", replace with "$\bar{␣}$", move left, and change to state $q_{\omega a\_ACCEPT\omega}$

In state $q_{\omega a\_ACCEPT\omega\_p\_\omega\{e\},1L\omega\_\omega\$,1L\omega}$:

- On any of the symbols ["0", "1", "$", "␣", "{<}", "{>}"], leave symbol unchanged, move left, and change to state $q_{\omega a\_ACCEPT\omega\_p\_\omega\{e\},1L\omega\_\omega\$,1L\omega}$
- On any of the symbols ["$\bar{0}$", "$\bar{1}$", "$\bar{\$}$", "$\bar{␣}$"], replace with "$", move left, and change to state $q_{\omega a\_ACCEPT\omega\_p\_\omega\{e\},1L\omega\_\omega\$,1L\omega\text{-}markHead}$

In state $q_{\omega a\_ACCEPT\omega\_p\_\omega\{e\},1L\omega\_\omega\$,1L\omega\text{-}markHead}$:

- On symbol "{<}", replace with "{<}", move right, and change to state $q_{\omega a\_ACCEPT\omega\_p\_\omega\{e\},1L\omega\_\omega\$,1L\omega\text{-}markHead}$
- On symbol "{>}", replace with "{$\bar{>}$}", move left, and change to state $q_{\omega a\_ACCEPT\omega\_p\_\omega\{e\},1L\omega}$
- On symbol "0", replace with "$\bar{0}$", move left, and change to state $q_{\omega a\_ACCEPT\omega\_p\_\omega\{e\},1L\omega}$
- On symbol "1", replace with "$\bar{1}$", move left, and change to state $q_{\omega a\_ACCEPT\omega\_p\_\omega\{e\},1L\omega}$
- On symbol "$", replace with "$\bar{\$}$", move left, and change to state $q_{\omega a\_ACCEPT\omega\_p\_\omega\{e\},1L\omega}$
- On symbol "␣", replace with "$\bar{␣}$", move left, and change to state $q_{\omega a\_ACCEPT\omega\_p\_\omega\{e\},1L\omega}$

In state $q_{\omega a\_REJECT\omega\_p\_\omega\{e\},1R\omega}$:

- On any of the symbols ["0", "1", "$", "␣", "{<}", "{>}"], leave symbol unchanged, move left, and change to state $q_{\omega a\_REJECT\omega\_p\_\omega\{e\},1R\omega}$
- On any of the symbols ["$\bar{0}$", "$\bar{1}$", "$\bar{\$}$", "$\bar{␣}$"], replace with "␣", move right, and change to state $q_{\omega a\_REJECT\omega\_p\_\omega\{e\},1R\omega\text{-}markHead}$

In state $q_{\omega a\_REJECT\omega\_p\_\omega\{e\},1R\omega\text{-}markHead}$:

- On symbol "{<}", replace with "{<}", move right, and change to state $q_{\omega a\_REJECT\omega\_p\_\omega\{e\},1R\omega\text{-}markHead}$
- On symbol "{>}", replace with "{$\bar{>}$}", move left, and change to state $q_{\omega a\_REJECT\omega}$
- On symbol "0", replace with "$\bar{0}$", move left, and change to state $q_{\omega a\_REJECT\omega}$
- On symbol "1", replace with "$\bar{1}$", move left, and change to state $q_{\omega a\_REJECT\omega}$
- On symbol "$", replace with "$\bar{\$}$", move left, and change to state $q_{\omega a\_REJECT\omega}$
- On symbol "␣", replace with "$\bar{␣}$", move left, and change to state $q_{\omega a\_REJECT\omega}$

In state $q_{\omega a\_REJECT\omega\_p\_\omega\{e\},1R\omega\_\omega\{e\},N\omega}$:

- On any of the symbols ["0", "1", "$", "␣", "{<}", "{>}"], leave symbol unchanged, move left, and change to state $q_{\omega a\_REJECT\omega\_p\_\omega\{e\},1R\omega\_\omega\{e\},N\omega}$
- On any of the symbols ["$\bar{0}$", "$\bar{1}$", "$\bar{\$}$", "$\bar{␣}$"], replace with "␣", do not move the tape head, and change to state $q_{\omega a\_REJECT\omega\_p\_\omega\{e\},1R\omega\_\omega\{e\},N\omega\text{-}markHead}$

In state $q_{\omega a\_REJECT\omega\_p\_\omega\{e\},1R\omega\_\omega\{e\},N\omega\text{-}markHead}$:

- On symbol "{<}", replace with "{<}", move right, and change to state $q_{\omega a\_REJECT\omega\_p\_\omega\{e\},1R\omega\_\omega\{e\},N\omega\text{-}markHead}$
- On symbol "{>}", replace with "{$\bar{>}$}", move left, and change to state $q_{\omega a\_REJECT\omega\_p\_\omega\{e\},1R\omega}$
- On symbol "0", replace with "$\bar{0}$", move left, and change to state $q_{\omega a\_REJECT\omega\_p\_\omega\{e\},1R\omega}$
- On symbol "1", replace with "$\bar{1}$", move left, and change to state $q_{\omega a\_REJECT\omega\_p\_\omega\{e\},1R\omega}$
- On symbol "$", replace with "$\bar{\$}$", move left, and change to state $q_{\omega a\_REJECT\omega\_p\_\omega\{e\},1R\omega}$
- On symbol "␣", replace with "$\bar{␣}$", move left, and change to state $q_{\omega a\_REJECT\omega\_p\_\omega\{e\},1R\omega}$

In state $q_{\omega a\_REJECT\omega}$:

- On symbol "{<}", replace with "{<}", move left, and change to state $q_{\omega a\_REJECT\omega\text{-}doRecord}$
- On any of the symbols ["0", "1", "$", "␣"], leave symbol unchanged, move left, and change to state $q_{\omega a\_REJECT\omega}$

In state $q_{\omega a\_REJECT\omega\text{-}doRecord}$:

- On symbol "␣", replace with "Q:{REJECT}", move right, and change to state $q_{\omega\_doEmptyCellInsertions}$

In state $q_{\omega\texttt{s\_qmore0s}\omega\texttt{\_f\_}\omega\texttt{0}\omega}$:
- On symbol "$\bar{\sqcup}$", replace with "$\bar{\sqcup}$", move right, and change to state $q_{\omega\texttt{a\_qmore0s}\omega\texttt{\_p\_}\omega\texttt{0,1R}\omega\texttt{\_}\omega\texttt{\{e\},1R}\omega}$
- On symbol "$\bar{\$}$", replace with "$\bar{\$}$", move right, and change to state $q_{\omega\texttt{a\_qmore0s}\omega\texttt{\_p\_}\omega\texttt{0,1R}\omega\texttt{\_}\omega\texttt{\$,1R}\omega}$
- On any of the symbols ["0", "1", "$", "␣", "{>}", "{<}"], leave symbol unchanged, move right, and change to state $q_{\omega\texttt{s\_qmore0s}\omega\texttt{\_f\_}\omega\texttt{0}\omega}$

In state $q_{\omega\texttt{a\_qmore0s}\omega\texttt{\_p\_}\omega\texttt{0,1R}\omega\texttt{\_}\omega\texttt{\{e\},1R}\omega}$:
- On any of the symbols ["0", "1", "$", "␣", "{<}", "{>}"], leave symbol unchanged, move left, and change to state $q_{\omega\texttt{a\_qmore0s}\omega\texttt{\_p\_}\omega\texttt{0,1R}\omega\texttt{\_}\omega\texttt{\{e\},1R}\omega}$
- On any of the symbols ["$\bar{0}$", "$\bar{1}$", "$\bar{\$}$", "$\bar{\text{␣}}$"], replace with "␣", move right, and change to state $q_{\omega\texttt{a\_qmore0s}\omega\texttt{\_p\_}\omega\texttt{0,1R}\omega\texttt{\_}\omega\texttt{\{e\},1R}\omega\texttt{-markHead}}$

In state $q_{\omega\texttt{a\_qmore0s}\omega\texttt{\_p\_}\omega\texttt{0,1R}\omega\texttt{\_}\omega\texttt{\{e\},1R}\omega\texttt{-markHead}}$:
- On symbol "{<}", replace with "{<}", move right, and change to state $q_{\omega\texttt{a\_qmore0s}\omega\texttt{\_p\_}\omega\texttt{0,1R}\omega\texttt{\_}\omega\texttt{\{e\},1R}\omega\texttt{-markHead}}$
- On symbol "{>}", replace with "{$\bar{>}$}", move left, and change to state $q_{\omega\texttt{a\_qmore0s}\omega\texttt{\_p\_}\omega\texttt{0,1R}\omega}$
- On symbol "0", replace with "$\bar{0}$", move left, and change to state $q_{\omega\texttt{a\_qmore0s}\omega\texttt{\_p\_}\omega\texttt{0,1R}\omega}$
- On symbol "1", replace with "$\bar{1}$", move left, and change to state $q_{\omega\texttt{a\_qmore0s}\omega\texttt{\_p\_}\omega\texttt{0,1R}\omega}$
- On symbol "$", replace with "$\bar{\$}$", move left, and change to state $q_{\omega\texttt{a\_qmore0s}\omega\texttt{\_p\_}\omega\texttt{0,1R}\omega}$
- On symbol "␣", replace with "$\bar{\text{␣}}$", move left, and change to state $q_{\omega\texttt{a\_qmore0s}\omega\texttt{\_p\_}\omega\texttt{0,1R}\omega}$

In state $q_{\omega\texttt{s\_qmore0s}\omega\texttt{\_f\_}\omega\texttt{1}\omega}$:
- On symbol "$\bar{\sqcup}$", replace with "$\bar{\sqcup}$", move right, and change to state $q_{\omega\texttt{a\_qmore0s}\omega\texttt{\_p\_}\omega\texttt{1,1R}\omega\texttt{\_}\omega\texttt{\{e\},1L}\omega}$
- On symbol "$\bar{\$}$", replace with "$\bar{\$}$", move right, and change to state $q_{\omega\texttt{a\_qmore1s}\omega\texttt{\_p\_}\omega\texttt{1,1R}\omega\texttt{\_}\omega\texttt{\$,1R}\omega}$
- On any of the symbols ["0", "1", "$", "␣", "{>}", "{<}"], leave symbol unchanged, move right, and change to state $q_{\omega\texttt{s\_qmore0s}\omega\texttt{\_f\_}\omega\texttt{1}\omega}$

In state $q_{\omega\texttt{a\_qmore0s}\omega\texttt{\_p\_}\omega\texttt{1,1R}\omega}$:
- On any of the symbols ["0", "1", "$", "␣", "{<}", "{>}"], leave symbol unchanged, move left, and change to state $q_{\omega\texttt{a\_qmore0s}\omega\texttt{\_p\_}\omega\texttt{1,1R}\omega}$
- On any of the symbols ["$\bar{0}$", "$\bar{1}$", "$\bar{\$}$", "$\bar{\text{␣}}$"], replace with "1", move right, and change to state $q_{\omega\texttt{a\_qmore0s}\omega\texttt{\_p\_}\omega\texttt{1,1R}\omega\texttt{-markHead}}$

In state $q_{\omega\texttt{a\_qmore0s}\omega\texttt{\_p\_}\omega\texttt{1,1R}\omega\texttt{-markHead}}$:
- On symbol "{<}", replace with "{<}", move right, and change to state $q_{\omega\texttt{a\_qmore0s}\omega\texttt{\_p\_}\omega\texttt{1,1R}\omega\texttt{-markHead}}$
- On symbol "{>}", replace with "{$\bar{>}$}", move left, and change to state $q_{\omega\texttt{a\_qmore0s}\omega}$
- On symbol "0", replace with "$\bar{0}$", move left, and change to state $q_{\omega\texttt{a\_qmore0s}\omega}$
- On symbol "1", replace with "$\bar{1}$", move left, and change to state $q_{\omega\texttt{a\_qmore0s}\omega}$
- On symbol "$", replace with "$\bar{\$}$", move left, and change to state $q_{\omega\texttt{a\_qmore0s}\omega}$
- On symbol "␣", replace with "$\bar{␣}$", move left, and change to state $q_{\omega\texttt{a\_qmore0s}\omega}$

In state $q_{\omega\texttt{a\_qmore0s}\omega\texttt{\_p\_}\omega\texttt{1,1R}\omega\texttt{\_}\omega\{\texttt{e}\}\texttt{,1L}\omega}$:
- On any of the symbols ["0", "1", "$", "␣", "{<}", "{>}"], leave symbol unchanged, move left, and change to state $q_{\omega\texttt{a\_qmore0s}\omega\texttt{\_p\_}\omega\texttt{1,1R}\omega\texttt{\_}\omega\{\texttt{e}\}\texttt{,1L}\omega}$
- On any of the symbols [$\bar{0}$, $\bar{1}$, $\bar{\$}$, "$\bar{␣}$"], replace with "␣", move left, and change to state $q_{\omega\texttt{a\_qmore0s}\omega\texttt{\_p\_}\omega\texttt{1,1R}\omega\texttt{\_}\omega\{\texttt{e}\}\texttt{,1L}\omega\texttt{-markHead}}$

In state $q_{\omega\texttt{a\_qmore0s}\omega\texttt{\_p\_}\omega\texttt{1,1R}\omega\texttt{\_}\omega\{\texttt{e}\}\texttt{,1L}\omega\texttt{-markHead}}$:
- On symbol "{<}", replace with "{<}", move right, and change to state $q_{\omega\texttt{a\_qmore0s}\omega\texttt{\_p\_}\omega\texttt{1,1R}\omega\texttt{\_}\omega\{\texttt{e}\}\texttt{,1L}\omega\texttt{-markHead}}$
- On symbol "{>}", replace with "{$\bar{>}$}", move left, and change to state $q_{\omega\texttt{a\_qmore0s}\omega\texttt{\_p\_}\omega\texttt{1,1R}\omega}$
- On symbol "0", replace with "$\bar{0}$", move left, and change to state $q_{\omega\texttt{a\_qmore0s}\omega\texttt{\_p\_}\omega\texttt{1,1R}\omega}$
- On symbol "1", replace with "$\bar{1}$", move left, and change to state $q_{\omega\texttt{a\_qmore0s}\omega\texttt{\_p\_}\omega\texttt{1,1R}\omega}$
- On symbol "$", replace with "$\bar{\$}$", move left, and change to state $q_{\omega\texttt{a\_qmore0s}\omega\texttt{\_p\_}\omega\texttt{1,1R}\omega}$
- On symbol "␣", replace with "$\bar{␣}$", move left, and change to state $q_{\omega\texttt{a\_qmore0s}\omega\texttt{\_p\_}\omega\texttt{1,1R}\omega}$

In state $q_{\omega\texttt{s\_qmore1s}\omega}$:
- On symbol "$\bar{␣}$", replace with "$\bar{␣}$", move right, and change to state $q_{\omega\texttt{s\_qmore1s}\omega\texttt{\_f\_}\omega\{\texttt{e}\}\omega}$
- On symbol "$\bar{1}$", replace with "$\bar{1}$", move right, and change to state $q_{\omega\texttt{s\_qmore1s}\omega\texttt{\_f\_}\omega\texttt{1}\omega}$
- On symbol "$\bar{0}$", replace with "$\bar{0}$", move right, and change to state $q_{\omega\texttt{s\_qmore1s}\omega\texttt{\_f\_}\omega\texttt{0}\omega}$
- On any of the symbols ["0", "1", "$", "␣", "{>}", "{<}"], leave symbol unchanged, move right, and change to state $q_{\omega\texttt{s\_qmore1s}\omega}$

In state $q_{\omega s\_qmore1s\omega\_f\_\omega\{e\}\omega}$:
- On symbol "$\bar{\$}$", replace with "$\bar{\$}$", move right, and change to state $q_{\omega a\_ACCEPT\omega\_p\_\omega\{e\},1L\omega\_\omega\$,1L\omega}$
- On symbol "$\bar{\sqcup}$", replace with "$\bar{\sqcup}$", move right, and change to state $q_{\omega a\_REJECT\omega\_p\_\omega\{e\},1R\omega\_\omega\{e\},N\omega}$
- On any of the symbols ["0", "1", "$", "␣", "{>}", "{<}"], leave symbol unchanged, move right, and change to state $q_{\omega s\_qmore1s\omega\_f\_\omega\{e\}\omega}$

In state $q_{\omega s\_qmore1s\omega\_f\_\omega1\omega}$:
- On symbol "$\bar{\sqcup}$", replace with "$\bar{\sqcup}$", move right, and change to state $q_{\omega a\_qmore1s\omega\_p\_\omega1,1R\omega\_\omega\{e\},1R\omega}$
- On symbol "$", replace with "$\bar{\$}$", move right, and change to state $q_{\omega a\_qmore1s\omega\_p\_\omega1,1R\omega\_\omega\$,1R\omega}$
- On any of the symbols ["0", "1", "$", "␣", "{>}", "{<}"], leave symbol unchanged, move right, and change to state $q_{\omega s\_qmore1s\omega\_f\_\omega1\omega}$

In state $q_{\omega a\_qmore1s\omega\_p\_\omega1,1R\omega\_\omega\{e\},1R\omega}$:
- On any of the symbols ["0", "1", "$", "␣", "{<}", "{>}"], leave symbol unchanged, move left, and change to state $q_{\omega a\_qmore1s\omega\_p\_\omega1,1R\omega\_\omega\{e\},1R\omega}$
- On any of the symbols ["$\bar{0}$", "$\bar{1}$", "$\bar{\$}$", "$\bar{\sqcup}$"], replace with "␣", move right, and change to state $q_{\omega a\_qmore1s\omega\_p\_\omega1,1R\omega\_\omega\{e\},1R\omega\text{-}markHead}$

In state $q_{\omega a\_qmore1s\omega\_p\_\omega1,1R\omega\_\omega\{e\},1R\omega\text{-}markHead}$:
- On symbol "{<}", replace with "{<}", move right, and change to state $q_{\omega a\_qmore1s\omega\_p\_\omega1,1R\omega\_\omega\{e\},1R\omega\text{-}markHead}$
- On symbol "{>}", replace with "{$\bar{>}$}", move left, and change to state $q_{\omega a\_qmore1s\omega\_p\_\omega1,1R\omega}$
- On symbol "0", replace with "$\bar{0}$", move left, and change to state $q_{\omega a\_qmore1s\omega\_p\_\omega1,1R\omega}$
- On symbol "1", replace with "$\bar{1}$", move left, and change to state $q_{\omega a\_qmore1s\omega\_p\_\omega1,1R\omega}$
- On symbol "$", replace with "$\bar{\$}$", move left, and change to state $q_{\omega a\_qmore1s\omega\_p\_\omega1,1R\omega}$
- On symbol "␣", replace with "$\bar{\sqcup}$", move left, and change to state $q_{\omega a\_qmore1s\omega\_p\_\omega1,1R\omega}$

In state $q_{\omega s\_qmore1s\omega\_f\_\omega0\omega}$:
- On symbol "$\bar{\sqcup}$", replace with "$\bar{\sqcup}$", move right, and change to state $q_{\omega a\_qmore1s\omega\_p\_\omega0,1R\omega\_\omega\{e\},1L\omega}$
- On symbol "$", replace with "$\bar{\$}$", move right, and change to state $q_{\omega a\_qmore0s\omega\_p\_\omega0,1R\omega\_\omega\$,1R\omega}$
- On any of the symbols ["0", "1", "$", "␣", "{>}", "{<}"], leave symbol unchanged, move right, and change to state $q_{\omega s\_qmore1s\omega\_f\_\omega0\omega}$

In state $q_{\omega a\_qmore1s\omega\_p\_\omega0,1R\omega}$:
- On any of the symbols ["0", "1", "$", "␣", "{<}", "{>}"], leave symbol unchanged, move left, and change to state $q_{\omega a\_qmore1s\omega\_p\_\omega0,1R\omega}$
- On any of the symbols ["$\bar{0}$", "$\bar{1}$", "$\bar{\$}$", "$\bar{␣}$"], replace with "0", move right, and change to state $q_{\omega a\_qmore1s\omega\_p\_\omega0,1R\omega\text{-}markHead}$

In state $q_{\omega a\_qmore1s\omega\_p\_\omega0,1R\omega\text{-}markHead}$:
- On symbol "{<}", replace with "{<}", move right, and change to state $q_{\omega a\_qmore1s\omega\_p\_\omega0,1R\omega\text{-}markHead}$
- On symbol "{>}", replace with "{$\bar{>}$}", move left, and change to state $q_{\omega a\_qmore1s\omega}$
- On symbol "0", replace with "$\bar{0}$", move left, and change to state $q_{\omega a\_qmore1s\omega}$
- On symbol "1", replace with "$\bar{1}$", move left, and change to state $q_{\omega a\_qmore1s\omega}$
- On symbol "$", replace with "$\bar{\$}$", move left, and change to state $q_{\omega a\_qmore1s\omega}$
- On symbol "␣", replace with "$\bar{␣}$", move left, and change to state $q_{\omega a\_qmore1s\omega}$

In state $q_{\omega a\_qmore1s\omega\_p\_\omega0,1R\omega\_\omega\{e\},1L\omega}$:
- On any of the symbols ["0", "1", "$", "␣", "{<}", "{>}"], leave symbol unchanged, move left, and change to state $q_{\omega a\_qmore1s\omega\_p\_\omega0,1R\omega\_\omega\{e\},1L\omega}$
- On any of the symbols ["$\bar{0}$", "$\bar{1}$", "$\bar{\$}$", "$\bar{␣}$"], replace with "␣", move left, and change to state $q_{\omega a\_qmore1s\omega\_p\_\omega0,1R\omega\_\omega\{e\},1L\omega\text{-}markHead}$

In state $q_{\omega a\_qmore1s\omega\_p\_\omega0,1R\omega\_\omega\{e\},1L\omega\text{-}markHead}$:
- On symbol "{<}", replace with "{<}", move right, and change to state $q_{\omega a\_qmore1s\omega\_p\_\omega0,1R\omega\_\omega\{e\},1L\omega\text{-}markHead}$
- On symbol "{>}", replace with "{$\bar{>}$}", move left, and change to state $q_{\omega a\_qmore1s\omega\_p\_\omega0,1R\omega}$
- On symbol "0", replace with "$\bar{0}$", move left, and change to state $q_{\omega a\_qmore1s\omega\_p\_\omega0,1R\omega}$
- On symbol "1", replace with "$\bar{1}$", move left, and change to state $q_{\omega a\_qmore1s\omega\_p\_\omega0,1R\omega}$
- On symbol "$", replace with "$\bar{\$}$", move left, and change to state $q_{\omega a\_qmore1s\omega\_p\_\omega0,1R\omega}$
- On symbol "␣", replace with "$\bar{␣}$", move left, and change to state $q_{\omega a\_qmore1s\omega\_p\_\omega0,1R\omega}$

```
TM.getState( "q_START" )
 .addTransitionGadget( WILDCARD, "FIRST-CELL-MARKER", "{^^}" ,
    "q_ω_init_insertLeftBound" )

TM.getState( "q_ω_init_insertLeftBound" )
 .addTransitionGadget( WILDCARD, "INSERT-AFTER-MARKER", "
    {^^}" , "{<}" , "q_ω_init_insertEmpty" )
```

```
TM.getState( "qω_init_insertEmpty" )
 .addTransitionGadget( WILDCARD, "INSERT-AFTER-MARKER", "
    {^^}" , EMPTY_CELL , "qω_init_findEnd" )

TM.getState( "qω_init_findEnd" )
 .addTransition( EMPTY_CELL , EMPTY_CELL , R, "qω_init_afterEnd"
    )
 .addTransition( WILDCARD, NO_CHANGE, R, "qω_init_findEnd" )

TM.getState( "qω_init_afterEnd" )
 .addTransition( EMPTY_CELL , "{>}" , R(4), "qω_init_afterLast" )

TM.getState( "qω_init_afterLast" )
 .addTransition( EMPTY_CELL , "{$$}" , L, "qω_init_atBoundR" )

TM.getState( "qω_init_atBoundR" )
 .addTransition( EMPTY_CELL , "{>}" , L, "qω_init_atExtraHead" )
 .addTransition( "{>}" , "{>}" , L, "qω_init_findStart" )

TM.getState( "qω_init_atExtraHead" )
 .addTransition( EMPTY_CELL , "␣" , L, "qω_init_atBoundL" )

TM.getState( "qω_init_atBoundL" )
 .addTransition( EMPTY_CELL , "{<}" , L, "qω_init_atBoundR" )

TM.getState( "qω_init_findStart" )
 .addTransition( "{^^}" , "{^^}" , R, "qω_init_atStart" )
 .addTransition( WILDCARD, NO_CHANGE, L, "qω_init_findStart" )

TM.getState( "qω_init_atStart" )
 .addTransition( EMPTY_CELL , "Q:{Run}" , R(2),
    "qω_init_markFirstHead" )

TM.getState( "qω_init_markFirstHead" )
 .addTransition( "0" , "0̄" , L(2), "qω_process" )
 .addTransition( "1" , "1̄" , L(2), "qω_process" )
 .addTransition( "$" , "$̄" , L(2), "qω_process" )
 .addTransition( EMPTY_CELL , "␣" , L(2), "qω_process" )

TM.getState( "qω_process" )
 .addTransition( "Q:{Run}" , EMPTY_CELL , R, "qωs_qRunω" )
 .addTransition( "Q:{more0s}" , EMPTY_CELL , R, "qωs_qmore0sω" )
 .addTransition( "Q:{more1s}" , EMPTY_CELL , R, "qωs_qmore1sω" )
```

```
TM.getState( "q_ω_doEmptyCellInsertions" )
 .addTransition( "{<}" , "{<}" , R, "q_ω_doEmptyCellInsertions-search" )

TM.getState( "q_ω_doEmptyCellInsertions-search" )
 .addTransition( "{$$}" , "{$$}" , L, "q_ω_doEmptyCellInsertions-done" )
 .addTransition( "{x̄}" , "{x̄}" , R, "q_ω_doEmptyCellInsertions-findToGrab" )
 .addTransition( ["0", "1", "$", "␣", "0̄", "1̄", "$̄", "␣̄", "{<}", "{>}"],
    NO_CHANGE, R, "q_ω_doEmptyCellInsertions-search" )

TM.getState( "q_ω_doEmptyCellInsertions-findToGrab" )
 .addTransition( "{$$}" , "{$$}" , N, "q_ω_doEmptyCellInsertions-foundToGrab"
    )
 .addTransition( ["0", "1", "$", "␣", "0̄", "1̄", "$̄", "␣̄", "{<}", "{>}",
    "{x̄}"], NO_CHANGE, R, "q_ω_doEmptyCellInsertions-findToGrab" )

TM.getState( "q_ω_doEmptyCellInsertions-foundToGrab" )
 .addTransitionGadget( WILDCARD, "INSERT-AFTER-MARKER", "{x̄}"
    , "{>}" , "q_ω_doEmptyCellInsertions-justInserted" )

TM.getState( "q_ω_doEmptyCellInsertions-justInserted" )
 .addTransition( ["{<}", "{$$}"], NO_CHANGE, L(2),
    "q_ω_doEmptyCellInsertions-nowMakeEmptyHead" )

TM.getState( "q_ω_doEmptyCellInsertions-nowMakeEmptyHead" )
 .addTransition( "{x̄}" , "␣̄" , L, "q_ω_doEmptyCellInsertions-done" )

TM.getState( "q_ω_doEmptyCellInsertions-done" )
 .addTransition( "{x̄}" , "{x̄}" , R, "q_ω_doEmptyCellInsertions-findToGrab" )
 .addTransition( "Q:{ACCEPT}" , "Q:{ACCEPT}" , N,
    STATE_ACCEPT )
 .addTransition( "Q:{REJECT}" , "Q:{REJECT}" , N,
    STATE_REJECT )
 .addTransition( "Q:{Run}" , "Q:{Run}" , N, "q_ω_process" )
 .addTransition( "Q:{more0s}" , "Q:{more0s}" , N, "q_ω_process" )
 .addTransition( "Q:{more1s}" , "Q:{more1s}" , N, "q_ω_process" )
 .addTransition( ["0", "1", "$", "␣", "0̄", "1̄", "$̄", "␣̄", "{<}", "{>}"],
    NO_CHANGE, L, "q_ω_doEmptyCellInsertions-done" )
```

```
TM.getState( "qωs_qRunω" )
 .addTransition( "␣̄" , "␣̄" , R, "qωs_qRunω_f_ω{e}ω" )
 .addTransition( "0̄" , "0̄" , R, "qωs_qRunω_f_ω0ω" )
 .addTransition( "1̄" , "1̄" , R, "qωs_qRunω_f_ω1ω" )
 .addTransition( ["0", "1", "$", "␣", "{>}", "{<}"], NO_CHANGE, R,
    "qωs_qRunω" )

TM.getState( "qωs_qRunω_f_ω{e}ω" )
 .addTransition( "␣̄" , "␣̄" , R, "qωa_ACCEPTω_p_ω{e},1Rω_ω$,1Lω" )
 .addTransition( ["0", "1", "$", "␣", "{>}", "{<}"], NO_CHANGE, R,
    "qωs_qRunω_f_ω{e}ω" )

TM.getState( "qωa_ACCEPTω_p_ω{e},1Rω" )
 .addTransition( ["0", "1", "$", "␣", "{<}", "{>}"], NO_CHANGE, L,
    "qωa_ACCEPTω_p_ω{e},1Rω" )
 .addTransition( ["0̄", "1̄", "$̄", "␣̄"], EMPTY_CELL , R,
    "qωa_ACCEPTω_p_ω{e},1Rω-markHead" )

TM.getState( "qωa_ACCEPTω_p_ω{e},1Rω-markHead" )
 .addTransition( "{<}" , "{<}" , R, "qωa_ACCEPTω_p_ω{e},1Rω-markHead" )
 .addTransition( "{>}" , "{>̄}" , L, "qωa_ACCEPTω" )
 .addTransition( "0" , "0̄" , L, "qωa_ACCEPTω" )
 .addTransition( "1" , "1̄" , L, "qωa_ACCEPTω" )
 .addTransition( "$" , "$̄" , L, "qωa_ACCEPTω" )
 .addTransition( EMPTY_CELL , "␣̄" , L, "qωa_ACCEPTω" )

TM.getState( "qωa_ACCEPTω_p_ω{e},1Rω_ω$,1Lω" )
 .addTransition( ["0", "1", "$", "␣", "{<}", "{>}"], NO_CHANGE, L,
    "qωa_ACCEPTω_p_ω{e},1Rω_ω$,1Lω" )
 .addTransition( ["0̄", "1̄", "$̄", "␣̄"], "$" , L,
    "qωa_ACCEPTω_p_ω{e},1Rω_ω$,1Lω-markHead" )

TM.getState( "qωa_ACCEPTω_p_ω{e},1Rω_ω$,1Lω-markHead" )
 .addTransition( "{<}" , "{<}" , R, "qωa_ACCEPTω_p_ω{e},1Rω_ω$,1Lω-markHead"
    )
 .addTransition( "{>}" , "{>̄}" , L, "qωa_ACCEPTω_p_ω{e},1Rω" )
 .addTransition( "0" , "0̄" , L, "qωa_ACCEPTω_p_ω{e},1Rω" )
 .addTransition( "1" , "1̄" , L, "qωa_ACCEPTω_p_ω{e},1Rω" )
 .addTransition( "$" , "$̄" , L, "qωa_ACCEPTω_p_ω{e},1Rω" )
 .addTransition( EMPTY_CELL , "␣̄" , L, "qωa_ACCEPTω_p_ω{e},1Rω" )
```

```
TM.getState( "qωa_ACCEPTω" )
 .addTransition( "{<}" , "{<}" , L, "qωa_ACCEPTω-doRecord" )
 .addTransition( ["0", "1", "$", "␣"], NO_CHANGE, L, "qωa_ACCEPTω" )

TM.getState( "qωa_ACCEPTω-doRecord" )
 .addTransition( EMPTY_CELL , "Q:{ACCEPT}" , R,
    "qω_doEmptyCellInsertions" )

TM.getState( "qωs_qRunω_f_ω0ω" )
 .addTransition( "␣̄" , "␣̄" , R, "qωa_qmore0sω_p_ω0,1Rω_ω$,1Rω" )
 .addTransition( ["0", "1", "$", "␣", "{>}", "{<}"], NO_CHANGE, R,
    "qωs_qRunω_f_ω0ω" )

TM.getState( "qωa_qmore0sω_p_ω0,1Rω" )
 .addTransition( ["0", "1", "$", "␣", "{<}", "{>}"], NO_CHANGE, L,
    "qωa_qmore0sω_p_ω0,1Rω" )
 .addTransition( ["0̄", "1̄", "$̄", "␣̄"], "0" , R, "qωa_qmore0sω_p_ω0,1Rω-markHead"
    )

TM.getState( "qωa_qmore0sω_p_ω0,1Rω-markHead" )
 .addTransition( "{<}" , "{<}" , R, "qωa_qmore0sω_p_ω0,1Rω-markHead" )
 .addTransition( "{>}" , "{>̄}" , L, "qωa_qmore0sω" )
 .addTransition( "0" , "0̄" , L, "qωa_qmore0sω" )
 .addTransition( "1" , "1̄" , L, "qωa_qmore0sω" )
 .addTransition( "$" , "$̄" , L, "qωa_qmore0sω" )
 .addTransition( EMPTY_CELL , "␣̄" , L, "qωa_qmore0sω" )

TM.getState( "qωa_qmore0sω_p_ω0,1Rω_ω$,1Rω" )
 .addTransition( ["0", "1", "$", "␣", "{<}", "{>}"], NO_CHANGE, L,
    "qωa_qmore0sω_p_ω0,1Rω_ω$,1Rω" )
 .addTransition( ["0̄", "1̄", "$̄", "␣̄"], "$" , R,
    "qωa_qmore0sω_p_ω0,1Rω_ω$,1Rω-markHead" )

TM.getState( "qωa_qmore0sω_p_ω0,1Rω_ω$,1Rω-markHead" )
 .addTransition( "{<}" , "{<}" , R, "qωa_qmore0sω_p_ω0,1Rω_ω$,1Rω-markHead"
    )
 .addTransition( "{>}" , "{>̄}" , L, "qωa_qmore0sω_p_ω0,1Rω" )
 .addTransition( "0" , "0̄" , L, "qωa_qmore0sω_p_ω0,1Rω" )
 .addTransition( "1" , "1̄" , L, "qωa_qmore0sω_p_ω0,1Rω" )
 .addTransition( "$" , "$̄" , L, "qωa_qmore0sω_p_ω0,1Rω" )
 .addTransition( EMPTY_CELL , "␣̄" , L, "qωa_qmore0sω_p_ω0,1Rω" )
```

```
TM.getState( "q_ωa_qmore0sω" )
 .addTransition( "{<}" , "{<}" , L, "q_ωa_qmore0sω-doRecord" )
 .addTransition( ["0", "1", "$", "␣"], NO_CHANGE, L, "q_ωa_qmore0sω" )

TM.getState( "q_ωa_qmore0sω-doRecord" )
 .addTransition( EMPTY_CELL , "Q:{more0s}" , R,
    "q_ω_doEmptyCellInsertions" )

TM.getState( "q_ωs_qRunω_f_ω1ω" )
 .addTransition( "␣̄" , "␣̄" , R, "q_ωa_qmore1sω_p_ω1,1Rω_ω$,1Rω" )
 .addTransition( ["0", "1", "$", "␣", "{>}", "{<}"], NO_CHANGE, R,
    "q_ωs_qRunω_f_ω1ω" )

TM.getState( "q_ωa_qmore1sω_p_ω1,1Rω" )
 .addTransition( ["0", "1", "$", "␣", "{<}", "{>}"], NO_CHANGE, L,
    "q_ωa_qmore1sω_p_ω1,1Rω" )
 .addTransition( ["0̄", "1̄", "$̄", "␣̄"], "1" , R, "q_ωa_qmore1sω_p_ω1,1Rω-markHead"
    )

TM.getState( "q_ωa_qmore1sω_p_ω1,1Rω-markHead" )
 .addTransition( "{<}" , "{<}" , R, "q_ωa_qmore1sω_p_ω1,1Rω-markHead" )
 .addTransition( "{>}" , "{>̄}" , L, "q_ωa_qmore1sω" )
 .addTransition( "0" , "0̄" , L, "q_ωa_qmore1sω" )
 .addTransition( "1" , "1̄" , L, "q_ωa_qmore1sω" )
 .addTransition( "$" , "$̄" , L, "q_ωa_qmore1sω" )
 .addTransition( EMPTY_CELL , "␣̄" , L, "q_ωa_qmore1sω" )

TM.getState( "q_ωa_qmore1sω_p_ω1,1Rω_ω$,1Rω" )
 .addTransition( ["0", "1", "$", "␣", "{<}", "{>}"], NO_CHANGE, L,
    "q_ωa_qmore1sω_p_ω1,1Rω_ω$,1Rω" )
 .addTransition( ["0̄", "1̄", "$̄", "␣̄"], "$" , R,
    "q_ωa_qmore1sω_p_ω1,1Rω_ω$,1Rω-markHead" )

TM.getState( "q_ωa_qmore1sω_p_ω1,1Rω_ω$,1Rω-markHead" )
 .addTransition( "{<}" , "{<}" , R, "q_ωa_qmore1sω_p_ω1,1Rω_ω$,1Rω-markHead"
    )
 .addTransition( "{>}" , "{>̄}" , L, "q_ωa_qmore1sω_p_ω1,1Rω" )
 .addTransition( "0" , "0̄" , L, "q_ωa_qmore1sω_p_ω1,1Rω" )
 .addTransition( "1" , "1̄" , L, "q_ωa_qmore1sω_p_ω1,1Rω" )
 .addTransition( "$" , "$̄" , L, "q_ωa_qmore1sω_p_ω1,1Rω" )
 .addTransition( EMPTY_CELL , "␣̄" , L, "q_ωa_qmore1sω_p_ω1,1Rω" )
```

```
TM.getState( "qωa_qmore1sω" )
 .addTransition( "{<}" , "{<}" , L, "qωa_qmore1sω-doRecord" )
 .addTransition( ["0", "1", "$", "␣"], NO_CHANGE, L, "qωa_qmore1sω" )

TM.getState( "qωa_qmore1sω-doRecord" )
 .addTransition( EMPTY_CELL , "Q:{more1s}" , R,
    "qω_doEmptyCellInsertions" )

TM.getState( "qωs_qmore0sω" )
 .addTransition( "␣" , "␣" , R, "qωs_qmore0sω_f_ω{e}ω" )
 .addTransition( "0̄" , "0̄" , R, "qωs_qmore0sω_f_ω0ω" )
 .addTransition( "1̄" , "1̄" , R, "qωs_qmore0sω_f_ω1ω" )
 .addTransition( ["0", "1", "$", "␣", "{>}", "{<}"], NO_CHANGE, R,
    "qωs_qmore0sω" )

TM.getState( "qωs_qmore0sω_f_ω{e}ω" )
 .addTransition( "$̄" , "$̄" , R, "qωa_ACCEPTω_p_ω{e},1Lω_ω$,1Lω" )
 .addTransition( "␣" , "␣" , R, "qωa_REJECTω_p_ω{e},1Rω_ω{e},Nω" )
 .addTransition( ["0", "1", "$", "␣", "{>}", "{<}"], NO_CHANGE, R,
    "qωs_qmore0sω_f_ω{e}ω" )

TM.getState( "qωa_ACCEPTω_p_ω{e},1Lω" )
 .addTransition( ["0", "1", "$", "␣", "{<}", "{>}"], NO_CHANGE, L,
    "qωa_ACCEPTω_p_ω{e},1Lω" )
 .addTransition( ["0̄", "1̄", "$̄", "␣"], EMPTY_CELL , L,
    "qωa_ACCEPTω_p_ω{e},1Lω-markHead" )

TM.getState( "qωa_ACCEPTω_p_ω{e},1Lω-markHead" )
 .addTransition( "{<}" , "{<}" , R, "qωa_ACCEPTω_p_ω{e},1Lω-markHead" )
 .addTransition( "{>}" , "{>̄}" , L, "qωa_ACCEPTω" )
 .addTransition( "0" , "0̄" , L, "qωa_ACCEPTω" )
 .addTransition( "1" , "1̄" , L, "qωa_ACCEPTω" )
 .addTransition( "$" , "$̄" , L, "qωa_ACCEPTω" )
 .addTransition( EMPTY_CELL , "␣" , L, "qωa_ACCEPTω" )

TM.getState( "qωa_ACCEPTω_p_ω{e},1Lω_ω$,1Lω" )
 .addTransition( ["0", "1", "$", "␣", "{<}", "{>}"], NO_CHANGE, L,
    "qωa_ACCEPTω_p_ω{e},1Lω_ω$,1Lω" )
 .addTransition( ["0̄", "1̄", "$̄", "␣"], "$" , L,
    "qωa_ACCEPTω_p_ω{e},1Lω_ω$,1Lω-markHead" )
```

```
TM.getState( "qωa_ACCEPTω_p_ω{e},1Lω_ω$,1Lω-markHead" )
 .addTransition( "{<}" , "{<}" , R, "qωa_ACCEPTω_p_ω{e},1Lω_ω$,1Lω-markHead"
    )
 .addTransition( "{>}" , "{>̄}" , L, "qωa_ACCEPTω_p_ω{e},1Lω" )
 .addTransition( "0" , "0̄" , L, "qωa_ACCEPTω_p_ω{e},1Lω" )
 .addTransition( "1" , "1̄" , L, "qωa_ACCEPTω_p_ω{e},1Lω" )
 .addTransition( "$" , "$̄" , L, "qωa_ACCEPTω_p_ω{e},1Lω" )
 .addTransition( EMPTY_CELL , "␣̄" , L, "qωa_ACCEPTω_p_ω{e},1Lω" )

TM.getState( "qωa_REJECTω_p_ω{e},1Rω" )
 .addTransition( ["0", "1", "$", "␣", "{<}", "{>}"], NO_CHANGE, L,
    "qωa_REJECTω_p_ω{e},1Rω" )
 .addTransition( ["0̄", "1̄", "$̄", "␣̄"], EMPTY_CELL , R,
    "qωa_REJECTω_p_ω{e},1Rω-markHead" )

TM.getState( "qωa_REJECTω_p_ω{e},1Rω-markHead" )
 .addTransition( "{<}" , "{<}" , R, "qωa_REJECTω_p_ω{e},1Rω-markHead" )
 .addTransition( "{>}" , "{>̄}" , L, "qωa_REJECTω" )
 .addTransition( "0" , "0̄" , L, "qωa_REJECTω" )
 .addTransition( "1" , "1̄" , L, "qωa_REJECTω" )
 .addTransition( "$" , "$̄" , L, "qωa_REJECTω" )
 .addTransition( EMPTY_CELL , "␣̄" , L, "qωa_REJECTω" )

TM.getState( "qωa_REJECTω_p_ω{e},1Rω_ω{e},Nω" )
 .addTransition( ["0", "1", "$", "␣", "{<}", "{>}"], NO_CHANGE, L,
    "qωa_REJECTω_p_ω{e},1Rω_ω{e},Nω" )
 .addTransition( ["0̄", "1̄", "$̄", "␣̄"], EMPTY_CELL , N,
    "qωa_REJECTω_p_ω{e},1Rω_ω{e},Nω-markHead" )

TM.getState( "qωa_REJECTω_p_ω{e},1Rω_ω{e},Nω-markHead" )
 .addTransition( "{<}" , "{<}" , R, "qωa_REJECTω_p_ω{e},1Rω_ω{e},Nω-markHead"
    )
 .addTransition( "{>}" , "{>̄}" , L, "qωa_REJECTω_p_ω{e},1Rω" )
 .addTransition( "0" , "0̄" , L, "qωa_REJECTω_p_ω{e},1Rω" )
 .addTransition( "1" , "1̄" , L, "qωa_REJECTω_p_ω{e},1Rω" )
 .addTransition( "$" , "$̄" , L, "qωa_REJECTω_p_ω{e},1Rω" )
 .addTransition( EMPTY_CELL , "␣̄" , L, "qωa_REJECTω_p_ω{e},1Rω" )

TM.getState( "qωa_REJECTω" )
 .addTransition( "{<}" , "{<}" , L, "qωa_REJECTω-doRecord" )
 .addTransition( ["0", "1", "$", "␣"], NO_CHANGE, L, "qωa_REJECTω" )
```

```
TM.getState( "q_ωa_REJECTω-doRecord" )
 .addTransition( EMPTY_CELL , "Q:{REJECT}" , R,
    "q_ω_doEmptyCellInsertions" )

TM.getState( "q_ωs_qmore0sω_f_ω0ω" )
 .addTransition( "␣̄" , "␣̄" , R, "q_ωa_qmore0sω_p_ω0,1Rω_ω{e},1Rω" )
 .addTransition( "$̄" , "$̄" , R, "q_ωa_qmore0sω_p_ω0,1Rω_ω$,1Rω" )
 .addTransition( ["0", "1", "$", "␣", "{>}", "{<}"], NO_CHANGE, R,
    "q_ωs_qmore0sω_f_ω0ω" )

TM.getState( "q_ωa_qmore0sω_p_ω0,1Rω_ω{e},1Rω" )
 .addTransition( ["0", "1", "$", "␣", "{<}", "{>}"], NO_CHANGE, L,
    "q_ωa_qmore0sω_p_ω0,1Rω_ω{e},1Rω" )
 .addTransition( ["0̄", "1̄", "$̄", "␣̄"], EMPTY_CELL , R,
    "q_ωa_qmore0sω_p_ω0,1Rω_ω{e},1Rω-markHead" )

TM.getState( "q_ωa_qmore0sω_p_ω0,1Rω_ω{e},1Rω-markHead" )
 .addTransition( "{<}" , "{<}" , R, "q_ωa_qmore0sω_p_ω0,1Rω_ω{e},1Rω-markHead"
    )
 .addTransition( "{>}" , "{>̄}" , L, "q_ωa_qmore0sω_p_ω0,1Rω" )
 .addTransition( "0" , "0̄" , L, "q_ωa_qmore0sω_p_ω0,1Rω" )
 .addTransition( "1" , "1̄" , L, "q_ωa_qmore0sω_p_ω0,1Rω" )
 .addTransition( "$" , "$̄" , L, "q_ωa_qmore0sω_p_ω0,1Rω" )
 .addTransition( EMPTY_CELL , "␣̄" , L, "q_ωa_qmore0sω_p_ω0,1Rω" )

TM.getState( "q_ωs_qmore0sω_f_ω1ω" )
 .addTransition( "␣̄" , "␣̄" , R, "q_ωa_qmore0sω_p_ω1,1Rω_ω{e},1Lω" )
 .addTransition( "$̄" , "$̄" , R, "q_ωa_qmore1sω_p_ω1,1Rω_ω$,1Rω" )
 .addTransition( ["0", "1", "$", "␣", "{>}", "{<}"], NO_CHANGE, R,
    "q_ωs_qmore0sω_f_ω1ω" )

TM.getState( "q_ωa_qmore0sω_p_ω1,1Rω" )
 .addTransition( ["0", "1", "$", "␣", "{<}", "{>}"], NO_CHANGE, L,
    "q_ωa_qmore0sω_p_ω1,1Rω" )
 .addTransition( ["0̄", "1̄", "$̄", "␣̄"], "1" , R, "q_ωa_qmore0sω_p_ω1,1Rω-markHead"
    )
```

```
TM.getState( "qωa_qmore0sω_p_ω1,1Rω-markHead" )
 .addTransition( "{<}" , "{<}" , R, "qωa_qmore0sω_p_ω1,1Rω-markHead" )
 .addTransition( "{>}" , "{>̄}" , L, "qωa_qmore0sω" )
 .addTransition( "0" , "0̄" , L, "qωa_qmore0sω" )
 .addTransition( "1" , "1̄" , L, "qωa_qmore0sω" )
 .addTransition( "$" , "$̄" , L, "qωa_qmore0sω" )
 .addTransition( EMPTY_CELL , "␣̄" , L, "qωa_qmore0sω" )

TM.getState( "qωa_qmore0sω_p_ω1,1Rω_ω{e},1Lω" )
 .addTransition( ["0", "1", "$", "␣", "{<}", "{>}"], NO_CHANGE, L,
    "qωa_qmore0sω_p_ω1,1Rω_ω{e},1Lω" )
 .addTransition( ["0̄", "1̄", "$̄", "␣̄"], EMPTY_CELL , L,
    "qωa_qmore0sω_p_ω1,1Rω_ω{e},1Lω-markHead" )

TM.getState( "qωa_qmore0sω_p_ω1,1Rω_ω{e},1Lω-markHead" )
 .addTransition( "{<}" , "{<}" , R, "qωa_qmore0sω_p_ω1,1Rω_ω{e},1Lω-markHead"
    )
 .addTransition( "{>}" , "{>̄}" , L, "qωa_qmore0sω_p_ω1,1Rω" )
 .addTransition( "0" , "0̄" , L, "qωa_qmore0sω_p_ω1,1Rω" )
 .addTransition( "1" , "1̄" , L, "qωa_qmore0sω_p_ω1,1Rω" )
 .addTransition( "$" , "$̄" , L, "qωa_qmore0sω_p_ω1,1Rω" )
 .addTransition( EMPTY_CELL , "␣̄" , L, "qωa_qmore0sω_p_ω1,1Rω" )

TM.getState( "qωs_qmore1sω" )
 .addTransition( "␣̄" , "␣̄" , R, "qωs_qmore1sω_f_ω{e}ω" )
 .addTransition( "1̄" , "1̄" , R, "qωs_qmore1sω_f_ω1ω" )
 .addTransition( "0̄" , "0̄" , R, "qωs_qmore1sω_f_ω0ω" )
 .addTransition( ["0", "1", "$", "␣", "{>}", "{<}"], NO_CHANGE, R,
    "qωs_qmore1sω" )

TM.getState( "qωs_qmore1sω_f_ω{e}ω" )
 .addTransition( "$̄" , "$̄" , R, "qωa_ACCEPTω_p_ω{e},1Lω_ω$,1Lω" )
 .addTransition( "␣̄" , "␣̄" , R, "qωa_REJECTω_p_ω{e},1Rω_ω{e},Nω" )
 .addTransition( ["0", "1", "$", "␣", "{>}", "{<}"], NO_CHANGE, R,
    "qωs_qmore1sω_f_ω{e}ω" )

TM.getState( "qωs_qmore1sω_f_ω1ω" )
 .addTransition( "␣̄" , "␣̄" , R, "qωa_qmore1sω_p_ω1,1Rω_ω{e},1Rω" )
 .addTransition( "$" , "$" , R, "qωa_qmore1sω_p_ω1,1Rω_ω$,1Rω" )
 .addTransition( ["0", "1", "$", "␣", "{>}", "{<}"], NO_CHANGE, R,
    "qωs_qmore1sω_f_ω1ω" )
```

```
TM.getState( "qωa_qmore1sω_p_ω1,1Rω_ω{e},1Rω" )
 .addTransition( ["0", "1", "$", "␣", "{<}", "{>}"], NO_CHANGE, L,
    "qωa_qmore1sω_p_ω1,1Rω_ω{e},1Rω" )
 .addTransition( ["0̄", "1̄", "$̄", "␣̄"], EMPTY_CELL , R,
    "qωa_qmore1sω_p_ω1,1Rω_ω{e},1Rω-markHead" )

TM.getState( "qωa_qmore1sω_p_ω1,1Rω_ω{e},1Rω-markHead" )
 .addTransition( "{<}" , "{<}" , R, "qωa_qmore1sω_p_ω1,1Rω_ω{e},1Rω-markHead"
    )
 .addTransition( "{>}" , "{>̄}" , L, "qωa_qmore1sω_p_ω1,1Rω" )
 .addTransition( "0" , "0̄" , L, "qωa_qmore1sω_p_ω1,1Rω" )
 .addTransition( "1" , "1̄" , L, "qωa_qmore1sω_p_ω1,1Rω" )
 .addTransition( "$" , "$̄" , L, "qωa_qmore1sω_p_ω1,1Rω" )
 .addTransition( EMPTY_CELL , "␣̄" , L, "qωa_qmore1sω_p_ω1,1Rω" )

TM.getState( "qωs_qmore1sω_f_ω0ω" )
 .addTransition( "␣̄" , "␣̄" , R, "qωa_qmore1sω_p_ω0,1Rω_ω{e},1Lω" )
 .addTransition( "$̄" , "$̄" , R, "qωa_qmore0sω_p_ω0,1Rω_ω$,1Rω" )
 .addTransition( ["0", "1", "$", "␣", "{>}", "{<}"], NO_CHANGE, R,
    "qωs_qmore1sω_f_ω0ω" )

TM.getState( "qωa_qmore1sω_p_ω0,1Rω" )
 .addTransition( ["0", "1", "$", "␣", "{<}", "{>}"], NO_CHANGE, L,
    "qωa_qmore1sω_p_ω0,1Rω" )
 .addTransition( ["0̄", "1̄", "$̄", "␣̄"], "0" , R, "qωa_qmore1sω_p_ω0,1Rω-markHead"
    )

TM.getState( "qωa_qmore1sω_p_ω0,1Rω-markHead" )
 .addTransition( "{<}" , "{<}" , R, "qωa_qmore1sω_p_ω0,1Rω-markHead" )
 .addTransition( "{>}" , "{>̄}" , L, "qωa_qmore1sω" )
 .addTransition( "0" , "0̄" , L, "qωa_qmore1sω" )
 .addTransition( "1" , "1̄" , L, "qωa_qmore1sω" )
 .addTransition( "$" , "$̄" , L, "qωa_qmore1sω" )
 .addTransition( EMPTY_CELL , "␣̄" , L, "qωa_qmore1sω" )

TM.getState( "qωa_qmore1sω_p_ω0,1Rω_ω{e},1Lω" )
 .addTransition( ["0", "1", "$", "␣", "{<}", "{>}"], NO_CHANGE, L,
    "qωa_qmore1sω_p_ω0,1Rω_ω{e},1Lω" )
 .addTransition( ["0̄", "1̄", "$̄", "␣̄"], EMPTY_CELL , L,
    "qωa_qmore1sω_p_ω0,1Rω_ω{e},1Lω-markHead" )
```

```
TM.getState( "qωa_qmore1sω_p_ω0,1Rω_ω{e},1Lω-markHead" )
 .addTransition( "{<}" , "{<}" , R, "qωa_qmore1sω_p_ω0,1Rω_ω{e},1Lω-markHead"
    )
 .addTransition( "{>}" , "{>̄}" , L, "qωa_qmore1sω_p_ω0,1Rω" )
 .addTransition( "0" , "0̄" , L, "qωa_qmore1sω_p_ω0,1Rω" )
 .addTransition( "1" , "1̄" , L, "qωa_qmore1sω_p_ω0,1Rω" )
 .addTransition( "$" , "$̄" , L, "qωa_qmore1sω_p_ω0,1Rω" )
 .addTransition( EMPTY_CELL , "␣̄" , L, "qωa_qmore1sω_p_ω0,1Rω" )
```

E Appendix: Brainfuck and P''

The language P'' was introduced by Böhm in 1964, and the version of Brainfuck that I implement (see next appendix), excluding input and output, can be directly reduced to P''. Since P'' has been more formally shown to be Turing-complete (Böhm 1964), Brainfuck can also be considered Turing complete. In this appendix, I first lay out the P'' language and then show how Brainfuck instructions and P'' instructions can be mapped to each other.

E (a) P'' language

P'' operates on a tape with an alphabet C of n symbols, where $n \geq 1$, as well as an extra "blank" symbol, denoted c_0. The symbols in C are numbered $\{c_1, \ldots, c_n\}$. The program is provided an infinitely long tape (to the left, i.e. the right side of the tape is bounded) with a finite number of symbols in C and then infinite blank symbols (c_0). Programs in P'' are a sequence of "words" defined with three rules (Böhm 1964, 189):

1. "R" and "λ" are valid words in P''.
2. If "α" and "β" are valid words separately, so is the combination "$\alpha\beta$".
3. If "α" is a valid word, so is "(α)".

and work as follows:

- "R" shifts the head of the tape to the right (if possible, since the right side is bounded).
- "λ" replaces the symbol currently at the head of the tape, c_i, with $c_{i+1 \bmod (n+1)}$ and then shifts the head of the tape to the left.
- "$\alpha\beta$" applies the function "α" and then applies the function "β".
- "(α)" means to apply function "α" as long as the current symbol is not c_0.

Böhm also defines "$(\alpha)^m$" as a shortcut to mean the application of "α" m different times (Böhm 1964, 190).

While Böhm works on a left-infinite tape, Brainfuck is done on a right-infinite tape. It should be obvious that simply switching the "left" and "right" uses in the definition of P'' does not change its computational abilities, and thus that the resulting language would still be Turing complete.[43] We can call the language that results in switching these uses the "flipped version" of P'', in contrast with the "original version." In the discussions of mapping Brainfuck to P'' and vice versa, "right" and "left" refer to the flipped versions, e.g. "R" is considered to move *left*.

[43] If this is not obvious, then consider "viewing" the tape on which P'' is executed from the "back", i.e. where "left" and "right" have the opposite meanings.

E (b) Mapping Brainfuck to P''

First, let us map Brainfuck to the flipped version of P''. While in and of itself this simply means that P'' can compute anything that Brainfuck can, and not the other way around, it provides more familiarity with P'' before we show that the equivalence works the other way, which proves that Brainfuck is Turing-complete. Here, we are omitting the input and output operations of Brainfuck. See section 4 (a) beginning on page 60 for the details of the Brainfuck language.

Brainfuck has six commands that do not deal with input or output: "+" to increment a cell, "-" to decrement it, ">" to move the head of the tape right, "<" to move the head of the tape left (if possible, since Brainfuck works on a right-infinite tape), and "[" and "]" for loops.

Given that P'' operates on the alphabet of symbols $\{c_0, c_1, \ldots, c_n\}$, where c_0 is considered a blank symbol that is used for the infinitely many symbols to the right of the current data, let $c_0 = 0$, $c_1 = 1$, and so on until $c_n = 255$. Thus, the alphabet is the numbers 0 through 255, where "blank" cells hold the value 0.

To implement an in-place increment ("+"), the corresponding P'' would be "λR". The "λ" increments the cell and shifts the head of the tape right. The "R" then moves the head of the tape back to the left. Thus, the head of the tape ends up in the same location where it started, and the symbol that had been there (c_i) was incremented with overflow (to $c_{i+1 \bmod (n+1)}$). This matches the semantics of the "+" Brainfuck operation.

To implement an in-place decrement ("-"), we can make use of the fact that incrementing a Brainfuck value (or incrementing in P'') overflows back to the start. Thus, the P'' for "-" would be "$(\lambda R)^{255}$", i.e. perform 255 increments, equivalent to a single decrement with underflow.

To implement "<", which moves the head of the tape to the left, is fairly simple: we can just use the "R" P'' instruction, which does the exact same thing, moving the head of the tape to the left (since we are using the "flipped" version of P'').

On the other hand, ">" is a bit harder to implement. The only instruction in P'' that moves the head of the tape right is "λ", which does so after incrementing the symbol currently at the head of the tape. To avoid this unwanted side-effect, we can first decrement that symbol. Thus, ">" can be implemented as "$(\lambda R)^{255}\lambda$", i.e. decrement once and then increment while moving right, resulting overall in no change to the value while the tape head moves in the desired direction.

Finally, "[" and "]" are perhaps the easiest. Since c_0 is represented with "0", "[" corresponds directly to "(" in P'', and "]" likewise corresponds to ")".

Accordingly, to convert the computation that a Brainfuck program performs into P'', perform the following replacements:

- Replace each "+" with "λR".

- Replace each "-" with "$(\lambda R)^{255}$", meaning "λR" repeated 255 times.
- Replace each "<" with "R".
- Replace each ">" with "$(\lambda R)^{255}\lambda$", meaning "λR" repeated 255 times followed by another "λ".
- Replace each "[" with "(".
- Replace each "]" with ")".

E (c) Mapping P'' to Brainfuck

The mapping from P'' to Brainfuck, on the other hand, is a lot simpler, and should be more understandable now that we have some familiarity with the P'' language.

The "R" command, which moves the head of the tape to the left (remember, we are dealing with a flipped version of P''), is equivalent to the Brainfuck command "<".

The "λ" command, which increments the cell and then moves the tape head to the right, is equivalent to the two Brainfuck commands "+>".

Just as we could replace Brainfuck's "[" and "]" with P''s "(" and ")", we can do the same in the other direction.

Accordingly, to convert the computation that a P''program performs into Brainfuck, perform the following replacements:

- Replace each "R" with ">".
- Replace each "λ" with "+>".
- Replace each "(" with "[".
- Replace each ")" with "]".

Thus, it should be clear that what Brainfuck can compute (i.e. Brainfuck without input/output features) is equivalent to P'', and vice-versa. The addition of input and output features does not reduce the computational functionality available in Brainfuck. Thus, this version of Brainfuck is Turing-complete, since P'' is Turing-complete (Böhm 1964).

F Appendix: Brainfuck operational semantics

Below, the operational semantics that I am implementing for Brainfuck are given, for a cell size of 256. The operational semantics are given in terms of the handling of different *expressions*, and thus do not reflect implementation details about how the jump commands are executed by the implementation. Let σ be the data array and ρ be the data pointer:

$$\textsc{Increment}\ \frac{v = \sigma[\rho] \qquad v \neq 255}{\langle \mathrm{INC}, \sigma, \rho\rangle \Downarrow \langle \sigma\{\rho \mapsto v+1\}, \rho\rangle}$$

$$\textsc{IncrementOverflow}\ \frac{\sigma[\rho] = 255}{\langle \mathrm{INC}, \sigma, \rho\rangle \Downarrow \langle \sigma\{\rho \mapsto 0\}, \rho\rangle}$$

$$\textsc{Decrement}\ \frac{v = \sigma[\rho] \qquad v \neq 0}{\langle \mathrm{DEC}, \sigma, \rho\rangle \Downarrow \langle \sigma\{\rho \mapsto v-1\}, \rho\rangle}$$

$$\textsc{DecrementUnderflow}\ \frac{\sigma[\rho] = 0}{\langle \mathrm{DEC}, \sigma, \rho\rangle \Downarrow \langle \sigma\{\rho \mapsto 255\}, \rho\rangle}$$

$$\textsc{Next}\ \frac{}{\langle \mathrm{NEXT}, \sigma, \rho\rangle \Downarrow \langle \sigma, \rho+1\rangle}$$

$$\textsc{Previous}\ \frac{\rho \neq 0}{\langle \mathrm{PREV}, \sigma, \rho\rangle \Downarrow \langle \sigma, \rho-1\rangle}$$

$$\textsc{PreviousStart}\ \frac{}{\langle \mathrm{PREV}, \sigma, 0\rangle \Downarrow \langle \sigma, 0\rangle}$$

$$\textsc{WhileFalse}\ \frac{\sigma[\rho] = 0}{\langle \mathrm{WHILE}(e), \sigma, \rho\rangle \Downarrow \langle \sigma, \rho\rangle}$$

$$\textsc{WhileTrue}\ \frac{\sigma[\rho] \neq 0 \qquad \langle e, \sigma, \rho\rangle \Downarrow \langle \sigma', \rho'\rangle \quad \langle \mathrm{WHILE}(e), \sigma', \rho'\rangle \Downarrow \langle \sigma'', \rho''\rangle}{\langle \mathrm{WHILE}(e), \sigma, \rho\rangle \Downarrow \langle \sigma'', \rho''\rangle}$$

$$\textsc{Input}\ \frac{\mathsf{c} \text{ is ASCII code of the next input character}}{\langle \mathrm{INPUT}, \sigma, \rho\rangle \Downarrow \langle \sigma\{\rho \mapsto c\}, \rho\rangle}$$

$$\textsc{InputEOF}\ \frac{\text{There is no next input character}}{\langle \mathrm{INPUT}, \sigma, \rho\rangle \Downarrow \langle \sigma\{\rho \mapsto 255\}, \rho\rangle}$$

$$\textsc{Output}\ \frac{c = \sigma[\rho]}{\langle \mathrm{OUTPUT}, \sigma, \rho\rangle \Downarrow \langle \sigma, \rho\rangle}$$
character c is output

G Appendix: Brainfuck implementation

Given that separating out the individual transitions for LaTeXto render them results in an excessively long program that is not particularly readable, instead of defining the transitions the way that I have for prior examples, I am instead including the raw JavaScript for the implementation.

```
const CELLS_PER_DATA = 2; // Number of data cells for a value
const ALL_HEX = [ ...'0123456789ABCDEF' ];
TM.setNumTapes( 5, 'qRun' );
TM.getState( 'qRun' ).addMTapeTransition(
    [ [...'+-<>[],.'], EMPTY_CELL, EMPTY_CELL, EMPTY_CELL,
       EMPTY_CELL ],
    [ NO_CHANGE, '$', '$', NO_CHANGE, NO_CHANGE ],
    [ N, R, N, R, N ],
    'qFindInput'
);
TM.getState( 'qFindInput' ).addMTapeTransition(
    [ [...'+-<>[],.'], EMPTY_CELL, [ '$', EMPTY_CELL ],
       EMPTY_CELL, EMPTY_CELL ],
    [ NO_CHANGE, EMPTY_CELL, NO_CHANGE, EMPTY_CELL, EMPTY_CELL
       ],
    [ R, N, R, N, N ],
    'qFindInput'
).addMTapeTransition(
    [ '!', EMPTY_CELL, EMPTY_CELL, EMPTY_CELL, EMPTY_CELL ],
    [ EMPTY_CELL, EMPTY_CELL, EMPTY_CELL, EMPTY_CELL,
       EMPTY_CELL ],
    [ R, N, R, N, N ],
    'qCopyInput'
).addMTapeTransition(
    [ EMPTY_CELL, EMPTY_CELL, EMPTY_CELL, EMPTY_CELL,
       EMPTY_CELL ],
    [ EMPTY_CELL, EMPTY_CELL, EMPTY_CELL, EMPTY_CELL,
       EMPTY_CELL ],
    // for tape 0 don't move so that qDoSetup can assume
       EMPTY_CELL
    [ N, N, N, N, N ],
    'qDoSetup'
);

ALL_HEX.forEach(
    char => TM.getState( 'qCopyInput' )
        .addMTapeTransition(
            [ char, EMPTY_CELL, EMPTY_CELL, EMPTY_CELL,
               EMPTY_CELL ],
            [ EMPTY_CELL, EMPTY_CELL, EMPTY_CELL, char,
               EMPTY_CELL ],
            [ R, N, R, R, N ],
            'qCopyInput'
        )
);
TM.getState( 'qCopyInput' ).addMTapeTransition(
```

```
    [ EMPTY_CELL, EMPTY_CELL, EMPTY_CELL, EMPTY_CELL,
       EMPTY_CELL ],
    [ EMPTY_CELL, EMPTY_CELL, EMPTY_CELL, EMPTY_CELL,
       EMPTY_CELL ],
    [ N, N, N, L, N ],
    'qResetInput'
);
TM.getState( 'qResetInput' ).addMTapeTransition(
    [ EMPTY_CELL, EMPTY_CELL, EMPTY_CELL, ALL_HEX, EMPTY_CELL
       ],
    [ EMPTY_CELL, EMPTY_CELL, EMPTY_CELL, NO_CHANGE, EMPTY_CELL
        ],
    [ N, N, N, L, N ],
    'qResetInput'
).addMTapeTransition(
    [ EMPTY_CELL, EMPTY_CELL, EMPTY_CELL, EMPTY_CELL,
       EMPTY_CELL ],
    [ EMPTY_CELL, EMPTY_CELL, EMPTY_CELL, EMPTY_CELL,
       EMPTY_CELL ],
    [ L, N, L, N, N ],
    'qDoSetup'
);

TM.getState( 'qDoSetup' )
    .addMTapeTransition(
        [ EMPTY_CELL, EMPTY_CELL, EMPTY_CELL, EMPTY_CELL,
           EMPTY_CELL ],
        [ EMPTY_CELL, ';', EMPTY_CELL, EMPTY_CELL, EMPTY_CELL
           ],
        [ N, R(CELLS_PER_DATA + 1), N, N, N ],
        'qWriteDataRight'
    );
TM.getState( 'qWriteDataRight' ).addMTapeTransition(
    [ EMPTY_CELL, EMPTY_CELL, EMPTY_CELL, EMPTY_CELL,
       EMPTY_CELL ],
    [ EMPTY_CELL, '/', EMPTY_CELL, EMPTY_CELL, EMPTY_CELL ],
    [ N, L, N, N, N ],
    'qWriteDataZero'
);
TM.getState( 'qWriteDataZero' ).addMTapeTransition(
    [ EMPTY_CELL, EMPTY_CELL, EMPTY_CELL, EMPTY_CELL,
       EMPTY_CELL ],
    [ EMPTY_CELL, '0', EMPTY_CELL, EMPTY_CELL, EMPTY_CELL ],
    [ N, L, N, N, N ],
    'qWriteDataZero'
).addMTapeTransition(
    [ EMPTY_CELL, ';', EMPTY_CELL, EMPTY_CELL, EMPTY_CELL ],
    [ EMPTY_CELL, ';', EMPTY_CELL, EMPTY_CELL, EMPTY_CELL ],
    [ N, N, N, N, N ],
    'qFindStart'
);
TM.getState( 'qFindStart' )
    .addMTapeTransition(
```

```
        [ [ EMPTY_CELL, ...'+-<>[],.'], ';', EMPTY_CELL,
           EMPTY_CELL, EMPTY_CELL ],
        [ NO_CHANGE, ';', EMPTY_CELL, EMPTY_CELL, EMPTY_CELL ],
        [ L, N, L, N, N ],
        'qFindStart'
    )
    .addMTapeTransition(
        [ [ ...'+-<>[],.'], ';', '$', EMPTY_CELL, EMPTY_CELL ],
        [ NO_CHANGE, ';', '$', EMPTY_CELL, EMPTY_CELL ],
        [ N, N, N, N, N ],
        'qProcess'
    );

TM.getState( 'qProcess' )
    .addMTapeTransition(
        [ [ '+', '-', '>', '[' ], ';', '$', EMPTY_CELL,
           EMPTY_CELL ],
        [ NO_CHANGE, ';', '$', EMPTY_CELL, EMPTY_CELL ],
        [ N, R, N, N, N ],
        'qBeforeProcess'
    )
    .addMTapeTransition(
        [ '<', ';', '$', EMPTY_CELL, EMPTY_CELL ],
        [ '<', ';', '$', EMPTY_CELL, EMPTY_CELL ],
        [ N, L, N, N, N ],
        'qMoveL'
    )
    .addMTapeTransition(
        [ '.', ';', '$', EMPTY_CELL, EMPTY_CELL ],
        [ '.', ';', '$', EMPTY_CELL, EMPTY_CELL ],
        [ N, R, N, N, N ],
        'qDoPrint'
    )
    .addMTapeTransition(
        [ ',', ';', '$', EMPTY_CELL, EMPTY_CELL ],
        [ ',', ';', '$', EMPTY_CELL, EMPTY_CELL ],
        [ N, R, N, R, N ],
        'qDoRead'
    )
    // UNCONDITIONAL JUMP BACK
    .addMTapeTransition(
        [ ']', ';', '$', EMPTY_CELL, EMPTY_CELL ],
        [ ']', ';', '$', EMPTY_CELL, EMPTY_CELL ],
        [ L, N, R, N, N ],
        'qJumpBack'
    )
    // Program end
    .addMTapeTransition(
        [ EMPTY_CELL, ';', '$', EMPTY_CELL, EMPTY_CELL ],
        [ EMPTY_CELL, ';', '$', EMPTY_CELL, EMPTY_CELL ],
        [ N, N, N, N, N ],
        STATE_ACCEPT
    );
```

```
// General utility: after an operation where the data pointer
   is left in
// the middle of a data cell, go back to the start of that cell
    and then
// go to process
TM.getState( 'qAfterProcess' )
    .addMTapeTransition(
        [ [...'+-[.,'], ALL_HEX, '$', EMPTY_CELL, EMPTY_CELL ],
        [ NO_CHANGE, NO_CHANGE, '$', EMPTY_CELL, EMPTY_CELL ],
        [ N, L, N, N, N ],
        'qAfterProcess'
    ).addMTapeTransition(
        [ [...'+-[.,'], ';', '$', EMPTY_CELL, EMPTY_CELL ],
        [ NO_CHANGE, ';', '$', EMPTY_CELL, EMPTY_CELL ],
        [ R, N, N, N, N ],
        'qProcess'
    );

// General utility: before an operation where the data pointer
   needs to start
// at the end of the data cell, find that end and then switch
   to perform the
// operation (except in the case of moving right where we might
    be done)
TM.getState( 'qBeforeProcess' )
    .addMTapeTransition(
        [ [ '+', '-', '>', '[' ], ALL_HEX, '$', EMPTY_CELL,
           EMPTY_CELL ],
        [ NO_CHANGE, NO_CHANGE, '$', EMPTY_CELL, EMPTY_CELL ],
        [ N, R, N, N, N ],
        'qBeforeProcess'
    )
    .addMTapeTransition(
        [ [ '+', '-' ], [ ';', '/' ], '$', EMPTY_CELL,
           EMPTY_CELL ],
        [ NO_CHANGE, NO_CHANGE, '$', EMPTY_CELL, EMPTY_CELL ],
        [ N, L, N, N, N ],
        'qDoMath'
    )
    // Move right: no extra handling needed
    .addMTapeTransition(
        [ '>', ';', '$', EMPTY_CELL, EMPTY_CELL ],
        [ '>', ';', '$', EMPTY_CELL, EMPTY_CELL ],
        [ R, N, N, N, N ],
        'qProcess'
    )
    // Move right: fill in new cell
    .addMTapeTransition(
        [ '>', '/', '$', EMPTY_CELL, EMPTY_CELL ],
        [ '>', ';', '$', EMPTY_CELL, EMPTY_CELL ],
        [ N, R(CELLS_PER_DATA + 1), N, N, N ],
        'qWriteCellBound'
    )
    .addMTapeTransition(
```

```
        [ '[', [ ';', '/' ], '$', EMPTY_CELL, EMPTY_CELL ],
        [ '[', NO_CHANGE, '$', EMPTY_CELL, EMPTY_CELL ],
        [ N, L, N, N, N ],
        'qCheckZero'
    );

// GROUP: ADDITION and SUBTRACTION
ALL_HEX.forEach(
    ( char, idx, arr ) => {
        // Addition; overflow on F->0
        const nextChar = ( char === 'F' ? '0' : arr[idx + 1] );
        const afterAddState = ( char === 'F' ? 'qDoMath' : '
           qAfterProcess' );
        TM.getState( 'qDoMath' ).addMTapeTransition(
            [ '+', char, '$', EMPTY_CELL, EMPTY_CELL ],
            [ '+', nextChar, '$', EMPTY_CELL, EMPTY_CELL ],
            [ N, L, N, N, N ],
            afterAddState
        );

        // Subtraction; underflow on 0->F
        const prevChar = ( char === '0' ? 'F' : arr[idx - 1] );
        const afterSubState = ( char === '0' ? 'qDoMath' : '
           qAfterProcess' );
        TM.getState( 'qDoMath' ).addMTapeTransition(
            [ '-', char, '$', EMPTY_CELL, EMPTY_CELL ],
            [ '-', prevChar, '$', EMPTY_CELL, EMPTY_CELL ],
            [ N, L, N, N, N ],
            afterSubState
        );
    }
);
// Overflow and underflow
TM.getState( 'qDoMath' ).addMTapeTransition(
    [ [ '+', '-' ], ';', '$', EMPTY_CELL, EMPTY_CELL ],
    [ NO_CHANGE, ';', '$', EMPTY_CELL, EMPTY_CELL ],
    [ N, N, N, N, N ],
    'qAfterProcess'
);

// GROUP: MOVE-R
TM.getState( 'qWriteCellBound' )
    .addMTapeTransition(
        [ '>', EMPTY_CELL, '$', EMPTY_CELL, EMPTY_CELL ],
        [ '>', '/', '$', EMPTY_CELL, EMPTY_CELL ],
        [ N, L, N, N, N ],
        'qFillNewCell'
    );
TM.getState( 'qFillNewCell' )
    .addMTapeTransition(
        [ '>', EMPTY_CELL, '$', EMPTY_CELL, EMPTY_CELL ],
        [ '>', '0', '$', EMPTY_CELL, EMPTY_CELL ],
        [ N, L, N, N, N ],
        'qFillNewCell'
```

```
    )
    .addMTapeTransition(
        [ '>', ';', '$', EMPTY_CELL, EMPTY_CELL ],
        [ '>', ';', '$', EMPTY_CELL, EMPTY_CELL ],
        [ R, N, N, N, N ],
        'qProcess'
    );

// GROUP: MOVE-L
TM.getState( 'qMoveL' )
    .addMTapeTransition(
        [ '<', ALL_HEX, '$', EMPTY_CELL, EMPTY_CELL ],
        [ '<', NO_CHANGE, '$', EMPTY_CELL, EMPTY_CELL ],
        [ N, L, N, N, N ],
        'qMoveL'
    )
    .addMTapeTransition(
        [ '<', ';', '$', EMPTY_CELL, EMPTY_CELL ],
        [ '<', ';', '$', EMPTY_CELL, EMPTY_CELL ],
        [ R, N, N, N, N ],
        'qProcess'
    )
    // Handle the first cell
    .addMTapeTransition(
        [ '<', '$', '$', EMPTY_CELL, EMPTY_CELL ],
        [ '<', '$', '$', EMPTY_CELL, EMPTY_CELL ],
        [ N, R, N, N, N ],
        'qMoveL'
    );

// GROUP: MAYBE-JUMP
TM.getState( 'qCheckZero' )
    .addMTapeTransition(
        [ '[', [...'123456789ABCDEF'], '$', EMPTY_CELL,
           EMPTY_CELL ],
        [ '[', NO_CHANGE, '$', EMPTY_CELL, EMPTY_CELL ],
        [ N, L, N, N, N ],
        'qAfterProcess' // not zero,  go to start of this data
           value and process
    )
    .addMTapeTransition(
        [ '[', '0', '$', EMPTY_CELL, EMPTY_CELL ],
        [ '[', '0', '$', EMPTY_CELL, EMPTY_CELL ],
        [ N, L, N, N, N ],
        'qCheckZero'
    )
    .addMTapeTransition(
        [ '[', ';', '$', EMPTY_CELL, EMPTY_CELL ],
        [ '[', ';', '$', EMPTY_CELL, EMPTY_CELL ],
        [ R, N, R, N, N ],
        'qJumpForward'
    );
TM.getState( 'qJumpForward' )
    .addMTapeTransition(
```

```
        [ [ ...'+-<>,.'], ';', EMPTY_CELL, EMPTY_CELL,
           EMPTY_CELL ],
        [ NO_CHANGE, ';', EMPTY_CELL, EMPTY_CELL, EMPTY_CELL ],
        [ R, N, N, N, N ],
        'qJumpForward'
    )
    .addMTapeTransition(
        [ '[', ';', EMPTY_CELL, EMPTY_CELL, EMPTY_CELL ],
        [ '[', ';', EMPTY_CELL, EMPTY_CELL, EMPTY_CELL ],
        [ R, N, R, N, N ],
        'qJumpForward'
    )
    .addMTapeTransition(
        [ ']', ';', EMPTY_CELL, EMPTY_CELL, EMPTY_CELL ],
        [ ']', ';', EMPTY_CELL, EMPTY_CELL, EMPTY_CELL ],
        [ R, N, L, N, N ],
        'qJumpForward'
    )
    // At the closing ] we went right on the command and now we
        finished and do
    // NOT move so that the subsequent command is processed
    .addMTapeTransition(
        [ [ ...'+-<>[],.'], ';', '$', EMPTY_CELL, EMPTY_CELL ],
        [ NO_CHANGE, ';', '$', EMPTY_CELL, EMPTY_CELL ],
        [ N, N, N, N, N ],
        'qProcess'
    );

// GROUP: UNCONDITIONAL JUMP BACK
TM.getState( 'qJumpBack' )
    .addMTapeTransition(
        [ [ ...'+-<>,.'], ';', EMPTY_CELL, EMPTY_CELL,
           EMPTY_CELL ],
        [ NO_CHANGE, ';', EMPTY_CELL, EMPTY_CELL, EMPTY_CELL ],
        [ L, N, N, N, N ],
        'qJumpBack'
    )
    .addMTapeTransition(
        [ ']', ';', EMPTY_CELL, EMPTY_CELL, EMPTY_CELL ],
        [ ']', ';', EMPTY_CELL, EMPTY_CELL, EMPTY_CELL ],
        [ L, N, R, N, N ],
        'qJumpBack'
    )
    .addMTapeTransition(
        [ '[', ';', EMPTY_CELL, EMPTY_CELL, EMPTY_CELL ],
        [ '[', ';', EMPTY_CELL, EMPTY_CELL, EMPTY_CELL ],
        [ L, N, L, N, N ],
        'qJumpBack'
    )
    // At the closing [ we went left on the command and now we
       finished and go
    // back right to process that [
    .addMTapeTransition(
        [ [ ...'+-<>[],.'], ';', '$', EMPTY_CELL, EMPTY_CELL ],
```

```
        [ NO_CHANGE, ';', '$', EMPTY_CELL, EMPTY_CELL ],
        [ R, N, N, N, N ],
        'qProcess'
    );

// GROUP: PRINTING
ALL_HEX.forEach(
    ( char ) => {
        TM.getState( 'qDoPrint' )
            .addMTapeTransition(
                [ '.', char, '$', EMPTY_CELL, EMPTY_CELL ],
                [ '.', char, '$', EMPTY_CELL, char ],
                [ N, R, N, N, R ],
                'qDoPrint'
            );
    }
)
TM.getState( 'qDoPrint' ).addMTapeTransition(
    [ '.', [ ';', '/'], '$', EMPTY_CELL, EMPTY_CELL ],
    [ '.', NO_CHANGE, '$', EMPTY_CELL, EMPTY_CELL ],
    [ N, L, N, N, N ],
    'qAfterProcess'
);

// GROUP: READING
TM.getState( 'qDoRead' )
    .addMTapeTransition(
        // When there is no more input to read fill in with 0xF
        [ ',', ALL_HEX, '$', EMPTY_CELL, EMPTY_CELL ],
        [ ',', 'F', '$', EMPTY_CELL, EMPTY_CELL ],
        [ N, R, N, N, N ],
        'qDoRead'
    )
    .addMTapeTransition(
        [ ',', [ ';', '/' ], '$', WILDCARD, EMPTY_CELL ],
        [ ',', NO_CHANGE, '$', NO_CHANGE, EMPTY_CELL ],
        [ N, L, N, L, N ],
        'qAfterProcess'
    )
ALL_HEX.forEach(
    ( char ) => {
        TM.getState( 'qDoRead' )
            .addMTapeTransition(
                [ ',', WILDCARD, '$', char, EMPTY_CELL ],
                [ ',', char, '$', EMPTY_CELL, EMPTY_CELL ],
                [ N, R, N, R, N ],
                'qDoRead'
            )
    }
);
```

H Appendix: Overview of the code

In this appendix, I provide a high-level overview of how the JavaScript code that I wrote to accompany this thesis works. To be clear, many of the classes and functions in the code are not discussed here, and more details can be found in the inline documentation. I also do not discuss details of how the code was used to semi-automatically generate the LaTeX used to render the various examples, or how the machine is displayed to the developer when it is run. I start by building up some of the classes used in the JavaScript, and end by explaining how they go together.

The resulting code can be tried out for a limited time on my Tufts EECS personal website at `https://www.eecs.tufts.edu/~dscher02/Thesis/Demo.html`. The code for the compiler is also attached to this thesis and can be run locally.

H (a) Cell content and movement

The simplest classes to understand are the `"CellContent"` and `"Movement"` classes, so let us begin by examining those.

The `"CellContent"` class generally represents the contents of a single cell on the tape of the Turing Machine. It holds two pieces of data - the actual content (usually a `"string"` or JavaScript `"Symbol"`) and whether to indicate that the content is the head of a simulated tape, since those cells are represented differently to indicate their status as the head of a simulated tape. Instances of `"CellContent"` are *immutable* - once created they cannot be changed. It is in this class that the logic mentioned in footnote n.13 on page 29 to prevent developer symbols from looking like the internal symbols is implemented. The `"CellContent"` class sometimes also holds an *array of multiple symbols* instead of just one - this is used for both storing multi-match transitions and multi-tape transitions when those are supported; in the first case the array represents the various symbol options to match, and in the second it represents the symbols at the heads of the various tapes.

There is also a subclass of `"CellContent"`, `"StateNameContent"`, that is not exposed to the Turing Machine developer. This class is used in the simulation of multi-tape machines to indicate what state should be simulated next; it has a dedicated class to prevent conflicts with any of the symbols that the developer can define transitions for.

The `"Movement"` class represents the specific movement of a tape head along a single tape, including potentially moving multiple cells, or not moving at all. The primary data of the class is simply a number to represent the movement, where positive numbers indicate movement to the right, negative numbers indicate movement to the left, and the value zero indicates not moving the head of the tape at all.[44] The magnitude of the value indicates how many

[44]For the sake of discussion here, zero is treated as neither positive nor negative; positive

cells to move the head of the tape.

In addition to its constructor, the "Movement" class has three static methods that return instances of the "Movement" class. "Movement.right(n)" returns an instance of the "Movement" class representing a move "n" cells to the right, "Movement.left(n)" returns an instance of the "Movement" class representing a move "n" cells to the left, and "Movement.none()" returns an instance of the "Movement" class representing not moving the head of the tape at all. In the pseudocode examples given above, "R" refers to the function "Movement.right", "L" refers to the function "Movement.left", and "N" refers to either the function "Movement.none" or its return value. The way that the code supports a transition where the movement is given as either "R" or "R(5)", for example, comes from the static method "Movement.normalize". If given a "Movement" instance, that method simpy returns it, but otherwise it assumes that the value given is a function and calls it with the value "1" as the parameter. Thus, "R" gets normalized to "R(1)", and similarly for movements to the left. This is also why "N" can refer to either the no-move function or its return value - if it is the function, it will get normalized to the return value.

H (b) Tape updates and transition results

Building off of the "CellContent" and "Movement" classes, we next turn to the "TapeUpdate" class. Instances of this class represent the changes to be made *on a single tape* as the result of a transition. Like "CellContent" and "Movement", instances of "TapeUpdate" are immutable. The default implementation holds a single "CellContent" and a single "Movement", representing the new symbol to write to the tape and the movement to perform on the tape. The "TapeUpdate" class can be considered a "smart" object - rather than providing a way to retrieve the data it stores, it exposes a method "applyToTape()" that, when given a "MachineTape" (discussed shortly), performs the necessary operations to apply itself to the tape.

Four subclasses of "TapeUpdate" exist:

- "NCTapeUpdate"
- "GadgetShiftUpdate"
- "GadgetFirstCellMarkerUpdate"
- "GadgetAfterMarkerUpdate"

The names reflect how they are used; each represents an update to a machine tape that does something different from just writing a symbol and moving the head of the tape. "NCTapeUpdate" does not write any symbol (the "NC" means "no change") and simply moves the head of the tape. "GadgetShiftUpdate" moves the current cell content to the right (leaving

numbers are greater than zero and negative numbers are less than zero.

the old cell blank) and then performs some additional movement on the tape. "GadgetFirstCellMarkerUpdate" moves a series of cells to the right to insert a new symbol at the "start" of the tape (which is either the real start, or an empty cell to the left of where the update first began running). Finally, "GadgetAfterMarkerUpdate" inserts a symbol after some specified marker, moving the subsequent cells to the right. Instances of these "TapeUpdate" subclasses are only used when the relevant feature is enabled. The three gadget classes are intended to improve performance, and are unlikely to reflect features that a real machine would have; they are implemented to behave exactly the same way as the simulated gadget would if the feature was disabled.

Making use of the "TapeUpdate" class (and its various subclasses) is the "TransitionResult" class. "TransitionResult"s are made up of a series of "TapeUpdate" instances (one for each *real* tape of the machine) and the name of the new state that the machine should be in. Like "TapeUpdate", the "TransitionResult" class is a "smart" object; it exposes a method named "applyToMachine()" that, when given a "RealMachine" (discussed shortly), applies the individual "TapeUpdate"s to the corresponding tapes of the machine and then sets the new state of the machine. However, *unlike* all of the classes discussed so far, instances of "TransitionResult" are *not* immutable. As discussed in footnote n.9 on page 15, the underlying value of the marker for internal state names, represented here in the documentation by "ω" occasionally needs to be changed as the developer adds new states, if those states conflict with this marker for internal state names. Whenever the value changes, all existing transitions for which the new state is such an internal state need to be updated to reflect the new name. Thus, "TransitionResult" exposes an "updateForOmega()" mtehod to allow such name changes.

H (c) Transitions for a single state

The "TransitionSet" class is used to collect all of the transitions that are defined *from* a state. Each "TransitionSet" is specific to a state, but is unaware of that state - it only gets accessed when defining or retrieving transitions for that state. Since new transitions get added to it, it is obviously not immutable, but transitions that are already defined cannot be removed.

Three public methods are provided for recording new transitions:

- "createSupportedMoveTransition()"
- "createSupportedMTapeMoveTransitions()"
- "addTransitionForTargets()"

Because the "TransitionSet" class does not have access to the Turing Machine to define new states, it does not implement the functionality of *simulating* multi-move, gadget, or multi-tape transitions. The only features that it simulates (if needed) are the support for multi-match transitions, no-change tran-

sitions, and wildcard transitions. The first method that is used to add transitions, "createSupportedMoveTransition()", defines transitions on single tape machine that use one of either "TapeUpdate" or "NCTapeUpdate" for the actual updates. The second, "createSupportedMTapeMoveTransitions()", is used to define transitions on a multi-tape machine that use the same two types of updates. Finally, "addTransitionForTargets()" is used to define transitions for a single-tape machine that use one of the gadget tape update classes. No support is provided for gadgets on multi-tape machines.

To identify the transition from a state on a given symbol, the class exposes the "getTransition()" method. Given an instance of "CellContent", the "TransitionSet" will search through the individual transitions, multi-match transitions (if enabled), and the wildcard transition (if enabled) to find the matching "TransitionResult" for the given target symbol, or "undefined" if no such transition is found. In the case of multiple real tapes, the "CellContent" parameter will actually contain an array of the contents at the head of each tape, but the implementation works the same.

"TransitionSet" also provides a method "updateForOmega()" that is used to call the method of the same name on each of the "TransitionResult"s that it holds, for updating state names.

H (d) Building the machine

When the developer is building up the desired Turing Machine, they make use of the "MachineSimulator" and "ProxyState" classes. The "TM" in the various code examples is an instance of the "MachineSimulator" class. The "MachineSimulator" class provides three methods for defining new transitions: "addTransition()" and "addTransitionGadget()", for single-tape transitions, and then "addMTapeTransition()" for multi-tape transitions. The first one, "addTransition()", defines a new transition for a single-taped Turing Machine using parameters equivalent to the tuple:

$$Q \times (\Gamma|\Gamma[]|\texttt{WILDCARD}) \times Q \times (\Gamma|\texttt{NO\_CHANGE}) \times \{L, R, N, L(v), R(v)\}$$

though the parameters are not in that order. The second method for single-taped transitions, "addTransitionGadget()", defines a new transition that uses one of the various "gadgets", and thus corresponds to the the tuple (again not in the same parameter order):

$$Q \times (\Gamma|\Gamma[]|\texttt{WILDCARD}) \times G(Q, ...)$$

Finally, "addMTapeTransition()" defines a new transition for a multi-taped Turing Machine using parameters equivalent to the tuple:

$$Q \times ((\Gamma|\Gamma[]|\texttt{WILDCARD})^n|\texttt{WILDCARD}) \times Q \times (\Gamma|\texttt{NO\_CHANGE})^n \times (\{L, R, N, L(v), R(v)\})^n$$

though again not in that exact order.

However, rather than requiring the developer to repeat the name of the starting state each time, the "ProxyState" class is provided. When the developer uses the "MachineSimulator" class method "getState()", it returns a "ProxyState" instance that provides the same three methods for defining transitions, just without needing to specify the name of the starting state each time. That "ProxyState" is merely a *proxy* for the underlying "MachineSimulator". Thus, program #3 from page 10

```
TM.getState( "q_START" )
 .addTransition( EMPTY_CELL , EMPTY_CELL , R,
    STATE_ACCEPT )
 .addTransition( ["1", "3", "5", "7", "9"], "1" , R, "q_START" )
 .addTransition( ["2", "4", "6", "8", "0"], NO_CHANGE, R, "q_START" )
```

is equivalent to a program

```
TM
 .addTransition( "q_START", EMPTY_CELL , EMPTY_CELL , R,
    STATE_ACCEPT )
 .addTransition( "q_START", ["1", "3", "5", "7", "9"], "1" , R, "q_START" )
 .addTransition( "q_START", ["2", "4", "6", "8", "0"], NO_CHANGE, R,
    "q_START" )
```

H (e) Running the machine

Finally, we turn to how the machine is run. After the developer finishes configuring their "MachineSimulator", they call "convertToRealMachine()" to get back a "RealMachine". This "RealMachine" represents the underlying machine that is running; it does not get a bunch of "TransitionSet" instances, but rather a callback function that represents the actual transition function. That function is given parameters $Q \times \Gamma$ (or, in the case of multiple tapes, $Q \times \Gamma^n$) and expects back a "TransitionResult" that it applies (by calling the "applyToMachine()" method).

The "RealMachine" class exposes a setter for the name of the current state ("setCurrentState()") as well as access to each of the underlying "MachineTape" instances. Those tapes are then modified via the individual "TapeUpdate" instances that the "TransitionResult" holds, most commonly using the method "setCurrentCellContent()" (which sets the "CellContent" at the current head of the tape) and the method "applyMove()" (which applies a "Movement" to the tape, including not moving left past the first cell and automatically adding extra empty cells to the end as needed).

H (f) Putting it all together

Using each of these classes, the workflow for the developer should be pretty clear. A "`MachineSimulator`" is created and then configured with the individual states and transitions that the developer wants the machine to have. If the developer tries to use a feature or gadget that is not enabled, the "`MachineSimulator`" will transparently convert that usage into a simulated version of the feature. If the developer uses a symbol that might conflict with one of the symbols used by the "`MachineSimulator`", the developer's symbol is rendered differently. On the other hand, if the developer wants to use a state name that might conflict with one of the states used by the "`MachineSimulator`", the "`MachineSimulator`" adjusts its marker of internal state names and replaces existing uses. When the developer is done setting up the machine, they can call "`convertToRealMachine()`" to get back a "`RealMachine`" instance that actually executes the configured Turing Machine.