But Can it Run DOOM?

Compiling High-Level Code for a Turing Machine

Morgan Saville

Contents

1 Introduction

It is often the goal of Computer Scientists to make our code run more efficiently. This, however, is not always the case. The game of *Code Golf* is a wonderful example. In this game, players must write their code using as few characters as possible. While this has the side-effect of reducing disk usage, this is seldom the point of the exercise. On the contrary, the intention is not to create fast code, or beautiful code, or memory-efficient code — the intention is to create fun code. For a programmer, there is an undeniable allure to making your job harder for seemingly no reason.

The joy that can be found in these self-imposed restrictions is what motivates the premise of this book. By the end, we will be able to describe fully-fledged algorithms using only the transition rules of a Turing machine.

Fortunately, this book will not simply be a detailed solution to a problem nobody wanted to solve. The principles we explore will generalise very well to any type of non-standard computing engine, and also to programming language design in general.

The main steps will be as follows:

1. Designing a machine code format which can encode Turing machine transition rules.
2. Creating a Turing machine simulator which can execute that machine code.
3. Building a human-readable pseudo-assembly language which can be converted into the machine code.
4. Making a compiler for our own high-level C-like language, which turns it into the pseudo-assembly.

Stringing it all together, the end goal is to be able to write high-level code, and have it run on a Turing machine.

We will be using C++14 to implement these steps. Basic knowledge of this language will definitely be helpful when following along. However, prior knowledge of compilers will not be necessary.

2 What is a Turing Machine?

2.1 Definition

A Turing machine is a theoretical machine first described by Alan Turing in 1936. This machine consists of a tape (a long line of adjacent cells) extending infinitely far in both directions. Each cell contains a symbol. The machine has a read/write head positioned over one of the cells, which is capable of determining which symbol is written in that cell, and is also able to replace it with a different symbol. The machine also has some internal state which can change over time.

This machine follows a particular finite set of so-called *transition rules*. Each rule has the format:

> If the machine is in state **X**, and the symbol under the read/write head is symbol **A**, then replace it with symbol **B**, move the read/write head exactly one cell to the left/right, and then set the state to state **Y**.

which can be expressed more concisely as:

$$\Delta(X, A) = (B, \leftarrow, Y)$$

or

$$\Delta(X, A) = (B, \rightarrow, Y)$$

depending on whether the read/write head should move to the left or right, respectively.

Consequently, this machine can carry out computations. The set of transition rules are like a program, and the initial symbols on the tape are like the input to that program. The machine starts in some pre-defined state, repeatedly applies whichever rule is appropriate, and once it reaches a situation in which none of its rules apply, it terminates leaving whichever symbols remain on the tape to serve as the output.

The incredible thing about this hypothetical machine is that it can actually compute any computable algorithm. If an algorithm can run on a modern computer, it can be simulated with a finite set of Turing machine transition rules.

It is with this theorem that Turing defined the set of all computable programs as the set of programs which can be executed on a Turing machine. As such, if some machine (or say, programming language) can be used to simulate a Turing machine, it then follows that it can also compute any computable algorithm.

This is the main reason why we can be confident that we can compile high-level code into Turing machine transition rules — any computable algorithm can be run on a Turing machine.

In order to clarify our goals, we can define what we mean by a Turing machine program mathematically.

$$
\begin{aligned}
&S := \{\text{possible states}\} \text{ such that } \mid S \mid \text{ is finite} \\
&A := \{\text{possible symbols}\} \text{ such that } \mid A \mid \text{ is finite} \\
&D := \{\leftarrow, \rightarrow\} \\
&s_0 \in S \\
\\
&\Delta : (S \times A) \nrightarrow (A \times D \times S)
\end{aligned}
$$

Here, S is the finite set of possible states that the Turing machine can be in. A (short for alphabet) is the finite set of symbols which can be written on a cell in the tape. The Turing machine initially starts in state s_0.

Δ, the set of rules, is a partial function between the sets shown above. The cell under the read/write head, the position of the read/write head, and the state of the machine are all updated repeatedly according to this function, until no further rule applies.

Note that we have not said anything about what the individual elements of S and A might be. In fact, these elements will depend on the program we are running (i.e., some programs might require more states than others, or a different alphabet. Regardless, both of these sets will always be finite).

We will have a lot of flexibility when implementing the Turing machine simulator. For example, our states could be represented by natural numbers (state 0, state 1, etc.) or by strings (`"start"`, `"ready"`, etc.). We will switch between these two representations when it becomes helpful to do so.

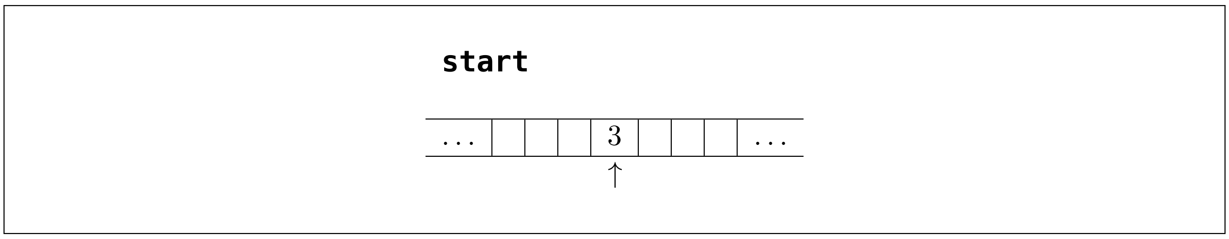
Similarly, we haven't specified what type of symbols we are using. When we actually come to implement the simulator, we will use strings to represent the symbols (most will only be a single character, but it will become very helpful to allow for multi-character symbols).

I should note that some may argue that A should only have exactly two elements (e.g., 0 and 1). This restriction doesn't really impact the computing power and would only serve to make the journey ahead much less enjoyable, so we will choose to ignore it.

2.2 Example 1: Output n of a given symbol

To really get a feel for how these machines operate, we will try to write a Turing machine program which takes in as input a digit from 0 to 3, and outputs the symbol `A` repeated that many times.

We will illustrate the operation of the machine at any given time using one of these diagrams:



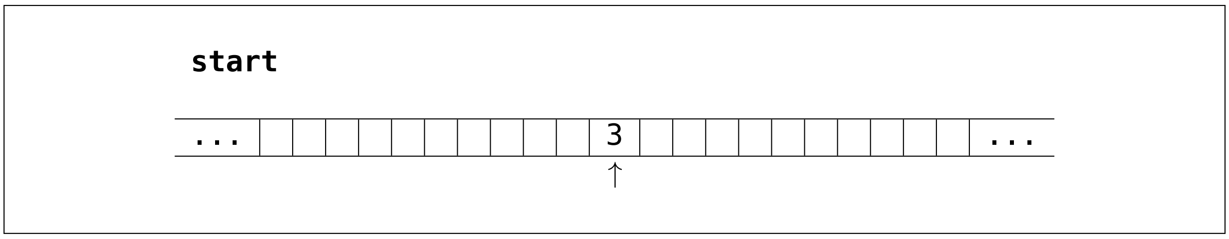
This diagram shows that the machine is in state **`start`** and the read/write head is over a cell which contains the symbol `3`, with the rest of the cells containing the empty symbol. This means that the next rule which will be executed corresponds to the transition $\Delta(\text{start}, 3)$, if such a transition exists in the ruleset.

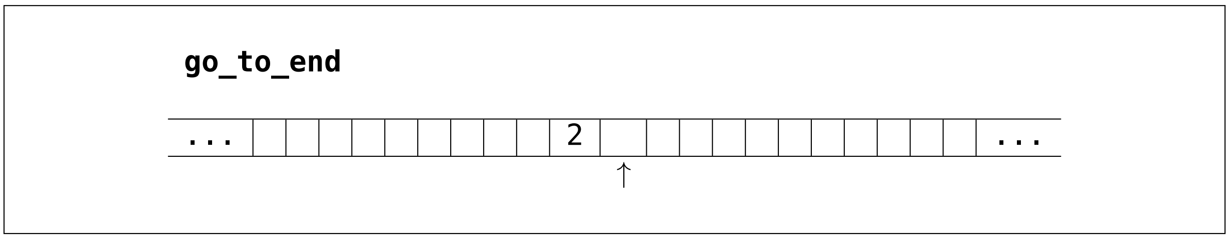
In fact, a ruleset which completes the task described above is the following:

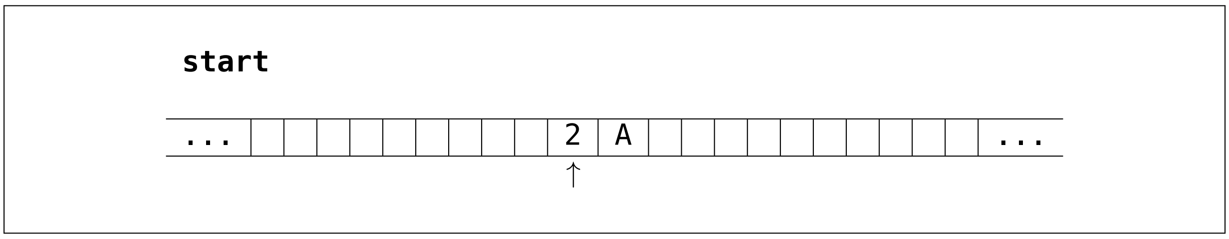
```
Δ(start, 0) = ( , →, done)
Δ(start, 1) = (0, →, go_to_end)
Δ(start, 2) = (1, →, go_to_end)
Δ(start, 3) = (2, →, go_to_end)
Δ(go_to_end, A) = (A, →, go_to_end)
Δ(go_to_end, ) = (A, ←, start)
Δ(start, A) = (A, ←, start)
```

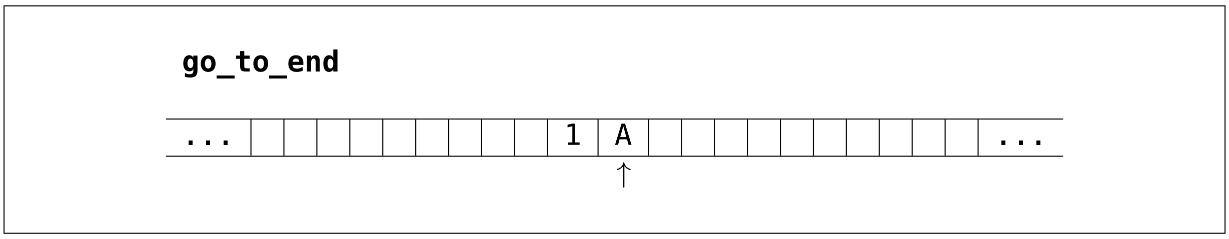
Remember that this is unlike a program in imperative programming languages like C, because the rules are not executed from top to bottom. We could shuffle the lines of this program randomly and the execution of the Turing machine would be indistinguishable.

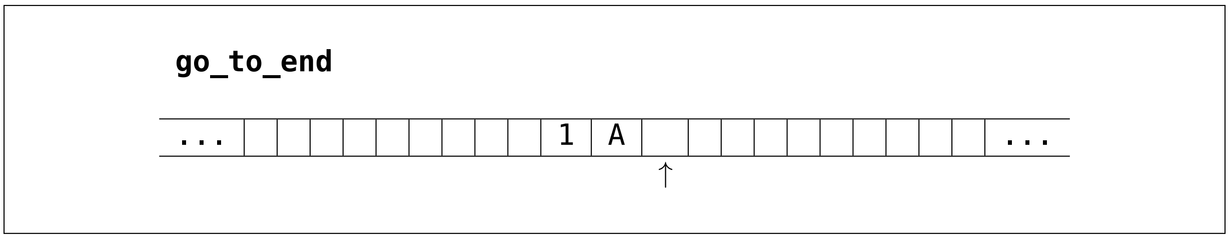
If we were to let the Turing machine run, we would see the following progression. Looking at the rules above, the jump from each diagram to the next should make sense.

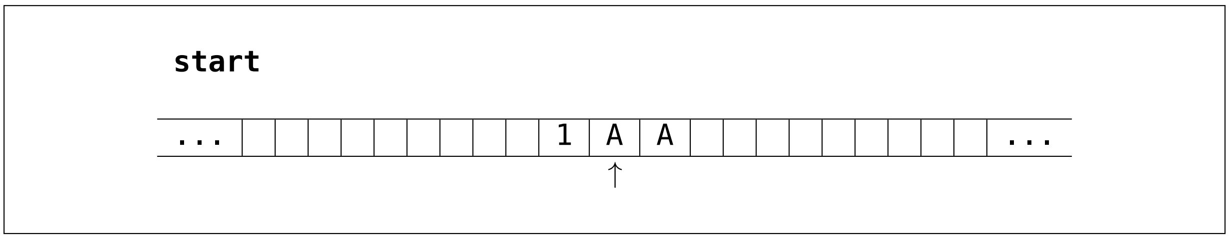

start
...
3
...

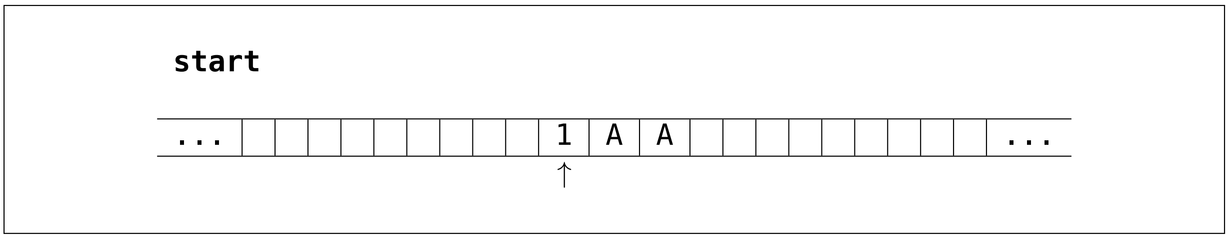

go_to_end
...
2
...

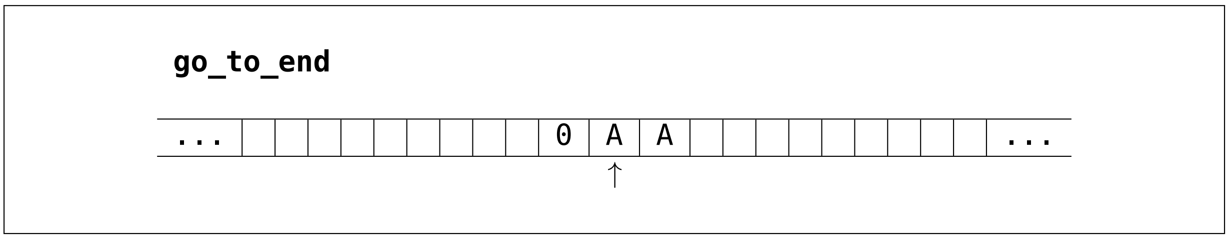

start
...
2
A
...

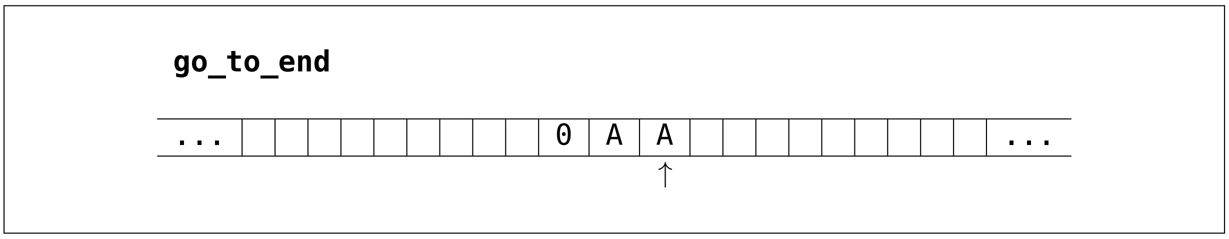

go_to_end
...
1
A
...

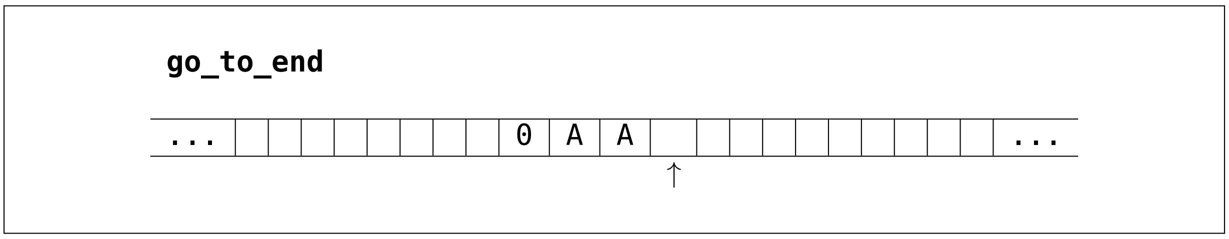

go_to_end
...
1
A
...

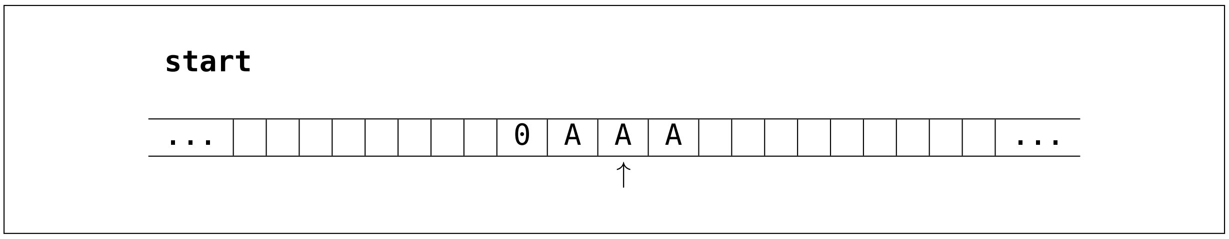

start
...
1
A
A
...

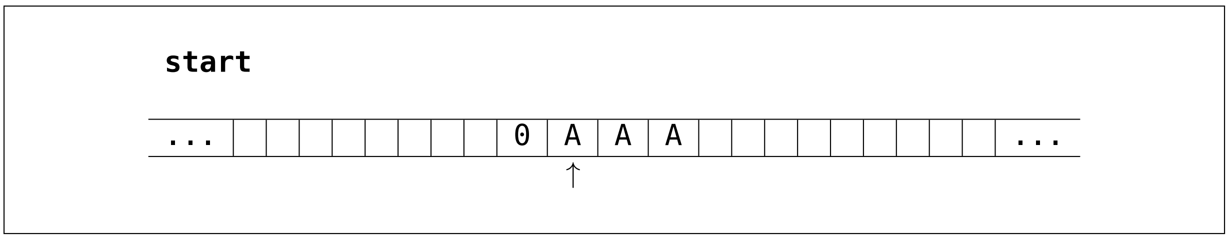

start
...
1
A
A
...

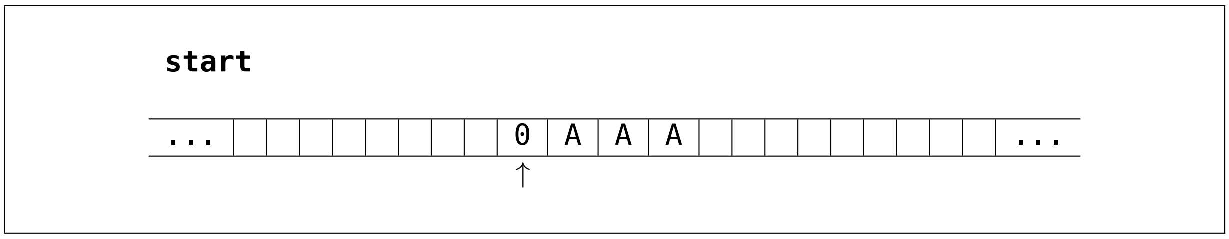

go_to_end
...
0
A
A
...

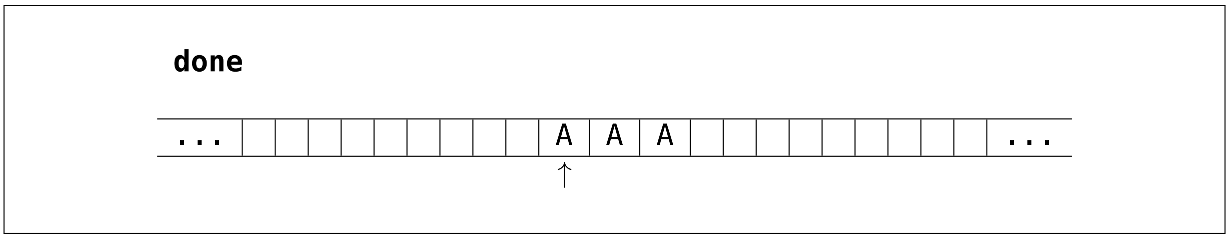

go_to_end
...
0
A
A
...


go_to_end
...
0
A
A
...


start
...
0
A
A
A
...


start
...
0
A
A
A
...





And it terminates here because there are no rules which tell the machine what to do when it is in state **done** and the current cell contains the symbol A.

As you can see, the program executed successfully and returned the expected output — a string of three As. It's worth taking some time to convince yourself that it would still work if the input was 2, 1, or 0.

It's also worth considering how we might extend the ruleset to accept larger inputs. Let's say we also wanted to be able to handle the input 4. Well, luckily, we can achieve this by only adding a single rule:

```
Δ(start, 0) = ( , →, done)
Δ(start, 1) = (0, →, go_to_end)
Δ(start, 2) = (1, →, go_to_end)
Δ(start, 3) = (2, →, go_to_end)
Δ(start, 4) = (3, →, go_to_end)
Δ(go_to_end, A) = (A, →, go_to_end)
Δ(go_to_end, ) = (A, ←, start)
Δ(start, A) = (A, ←, start)
```

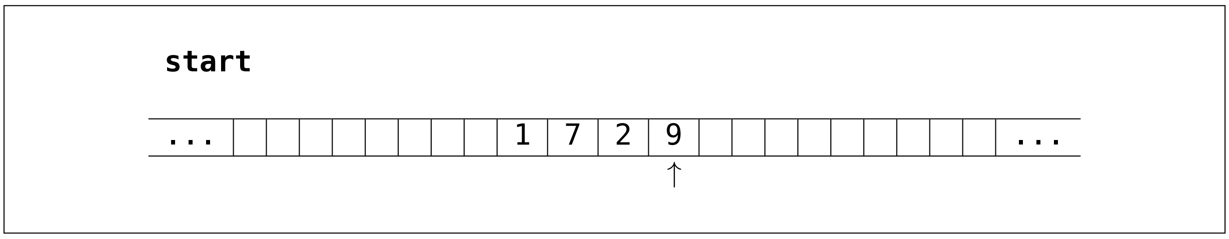
Similarly, we can handle inputs up to 9 with this method:

```
Δ(start, 0) = ( , →, done)
Δ(start, 1) = (0, →, go_to_end)
Δ(start, 2) = (1, →, go_to_end)
Δ(start, 3) = (2, →, go_to_end)
Δ(start, 4) = (3, →, go_to_end)
Δ(start, 5) = (4, →, go_to_end)
Δ(start, 6) = (5, →, go_to_end)
Δ(start, 7) = (6, →, go_to_end)
```

```
Δ(start, 8) = (7, →, go_to_end)
Δ(start, 9) = (8, →, go_to_end)
Δ(go_to_end, A) = (A, →, go_to_end)
Δ(go_to_end, ) = (A, ←, start)
Δ(start, A) = (A, ←, start)
```

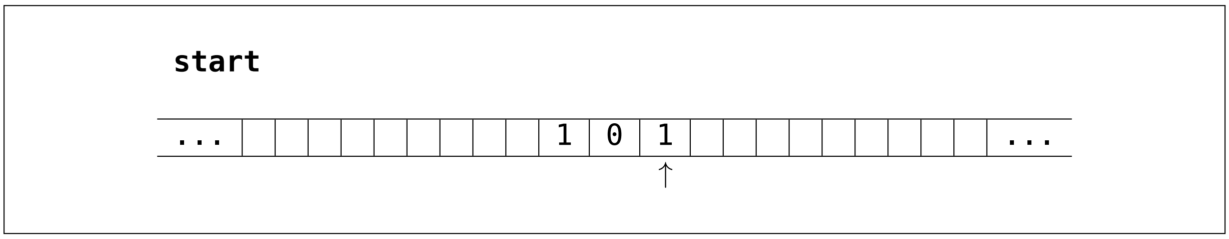
However, what if we want to handle arbitrarily large inputs? We could of course continue in the same manner, handling the two-character symbols (each on a single cell) `10` and `11` and so on, but it's clear that this is just pushing the problem back. To handle arbitrarily large inputs in this way, we would need infinitely many rules and an infinite set of symbols.

Instead, we will have to enable our program to handle having the input spread out across multiple cells. For example, if the input were 1729, the Turing machine would start in the following situation:



Note that we've decided to start with the read/write head over the rightmost digit of the input, but this is arbitrary. Now we only need to handle finitely many symbols — the digits `0`-`9`, the letter `A`, and the empty symbol.

For the sake of making the program easier to write manually, let's also say that the program works in binary, so for example if the input were 5 (which is written 101 in binary), then the Turing machine would start off as:



Now we can look at the program we used before and try to generalise it for multi-cell inputs. The three states we used were **`start`**, **`go_to_end`**, and **`done`**. The **`start`** state was used in three cases:

- When the read/write head is over a non-zero digit, and must decrement it,

- When the read/write head is over a zero digit, and must terminate,
- When the read/write head is somewhere in the list of `A`s and must return back to check the input digit.

We can use the same state for all of these cases because there is no overlap between the symbols which will be encountered in them.

The **`go_to_end`** state was used for when the read/write head is travelling to the right-hand-side of the list of `A`s, and for dropping an additional `A` when it gets there.

The **`done`** state, trivially, is for when the program has terminated.

For the sake of brevity and clarity, I may use anthropomorphism and metonymy to describe the Turing machine, using phrases such as "we are in state **`X`**" to mean "the Turing machine is in state **`X`**". Furthermore, I might use phrases such as "we see symbol `a`", "the machine sees symbol `a`", or "state **`X`** sees symbol `a`" all to mean that the read/write head is currently over a cell containing the symbol `a`. Similarly, I may say that in a given situation, we "want to find" a symbol, or even that a particular state "wants to find" a symbol. By this I mean that when the Turing machine is in that state, it should stay in that state and continue to move in the same direction, leaving all encountered symbols unchanged except for the one it "wants to find". Hopefully this type of shorthand is easily understood, and makes the logic behind our programs even clearer. However, if you find yourself getting lost in the abstractions it might be helpful to examine these metaphors in the context of what the machine itself is actually doing.

With this in mind, a version of our program which can handle multi-cell inputs might look as follows (remembering that we are working in binary):

> If we are in state **`start`** and we see a `1`, decrement it to a `0`, move to the right, and transition into the **`go_to_end`** state.
>
> If we are in state **`start`** and we see a `0`, we can replace it with a `1` and move to the left, staying in the **`start`** state. This is because when we decrement the number, the `0` rolls over to a `1` and we must similarly decrement the digit to its left.

With these two rules, two cases may occur:

- The input number was non-zero, and it has now been decremented. The read/write head is somewhere in the middle of it, or on the space immediately after it. The new state is **`go_to_end`**.
- The input number was zero, in which case it has been set to a string of `1` symbols. The read/write head is in the empty cell immediately before this string, and the state is still **`start`**

We can handle the latter case fairly simply with the following rules:

> If we are in state **`start`** and we see the empty symbol, we keep it empty, move to the right, and go to state **`done`**.

> If we are in state **`done`** and we see a `1`, we replace it with the empty symbol, move to the right, and stay in state **`done`**.

Here, the **`done`** state is used to clean up that string of `1`s, and the program terminates when there are no more to clean up.

Now we can handle the other case. We need to travel to the end of the input string, and also of the string of `A`s we've placed so far. Once we've gotten there, drop an `A`, turn back left, and go back to the input number to start again.

> If we are in state **`go_to_end`** and we see a `0`, `1`, or `A`, leave it as it is, move to the right, and stay in state **`go_to_end`**.

> If we are in state **`go_to_end`** and we see an empty cell, replace it with an `A`, move left, and go to state **`start`**.

These are the same rules which governed the **`go_to_end`** state in the single-cell-input example (except that now this state might encounter a `0` or `1`). As with that example, we need a rule which lets a machine in the **`start`** state which is somewhere in the string of `A`s find its way back to the input number.

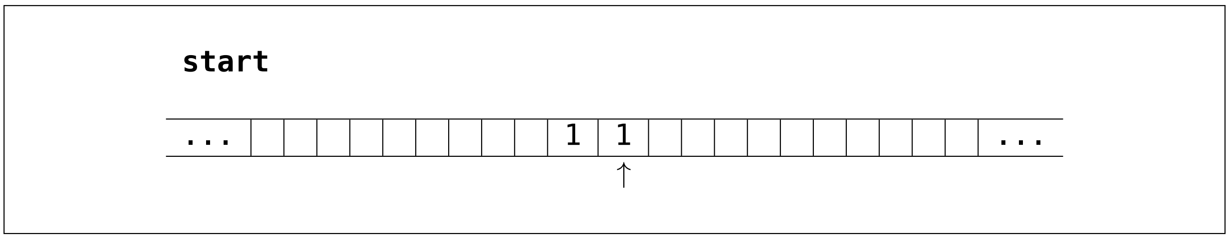
> If we are in state **`start`** and we see an `A`, then we leave it as an `A`, go to the left, and remain in state **`start`**.

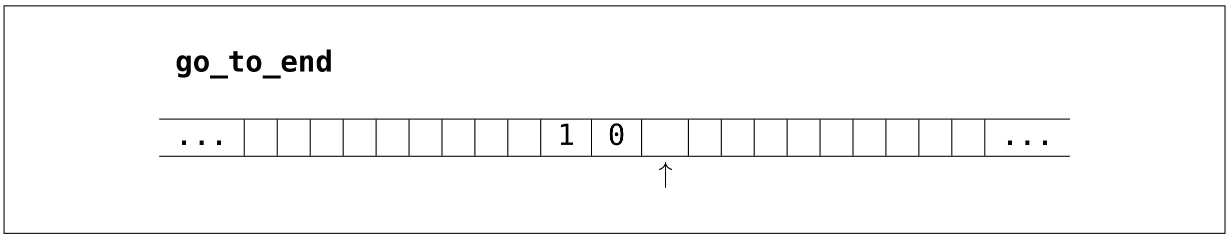
Eventually, the read/write head will find its way back to the rightmost digit of the input number, at which point one of our first two rules will be executed, and the cycle continues until the program terminates.

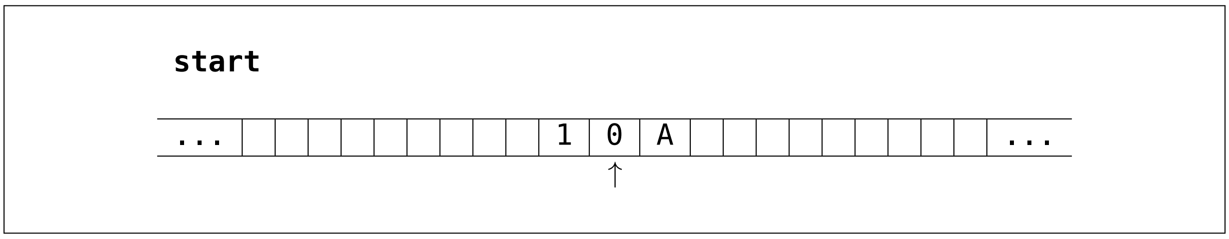
All of these rules can be expressed using the Δ function notation as follows:

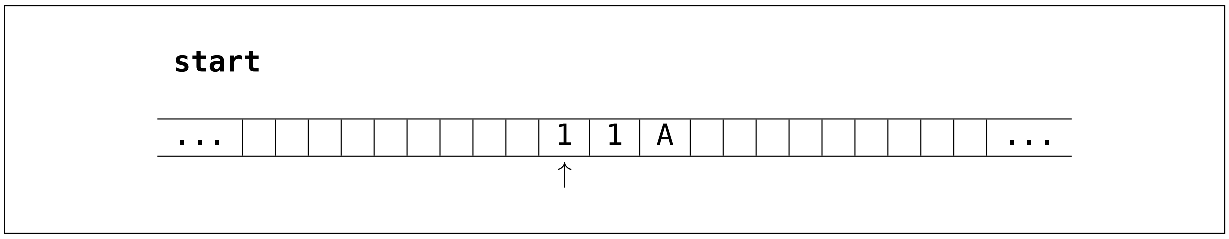
```
Δ(start, 1) = (0, →, go_to_end)
Δ(start, 0) = (1, ←, start)
Δ(start, ) = (, →, done)
Δ(done, 1) = (, →, done)
Δ(go_to_end, 0) = (0, →, go_to_end)
Δ(go_to_end, 1) = (1, →, go_to_end)
Δ(go_to_end, A) = (A, →, go_to_end)
Δ(go_to_end, ) = (A, ←, start)
Δ(start, A) = (A, ←, start)
```

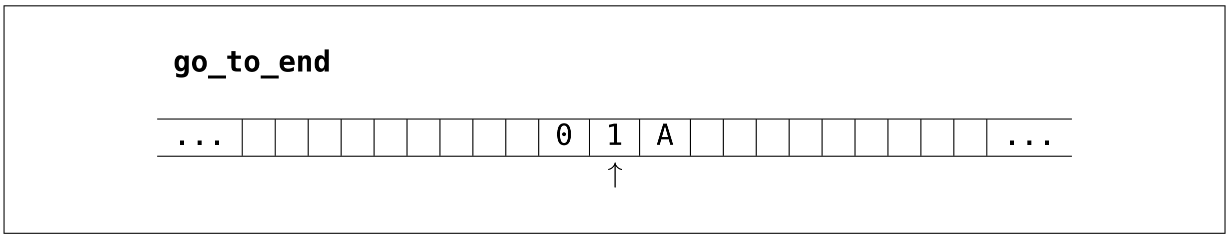
With an input of 3 (11 in binary), the program will run in the following way:

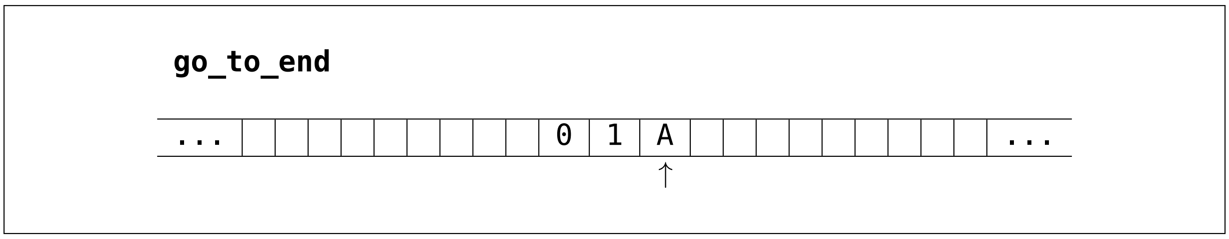


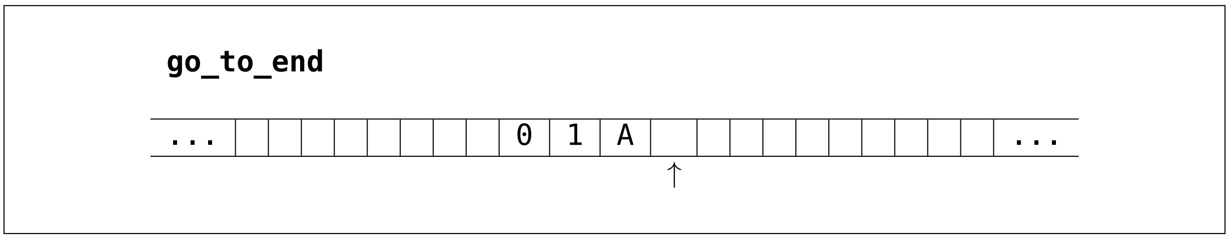


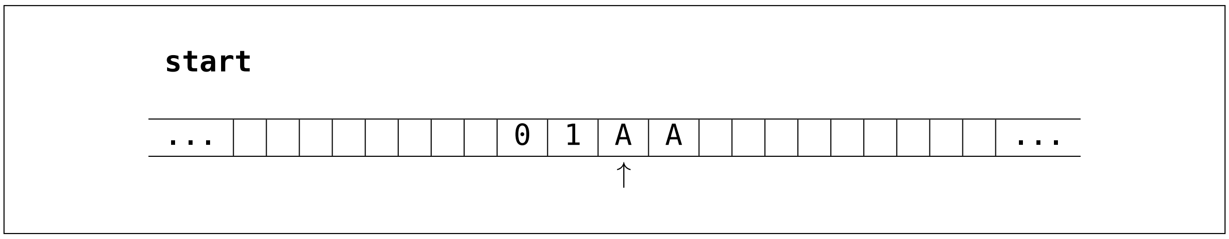


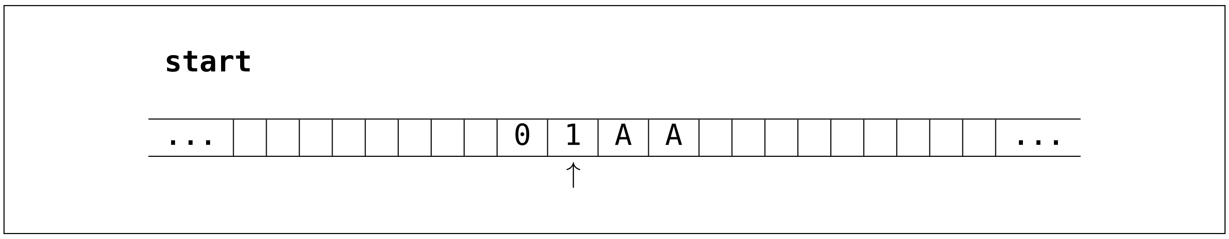


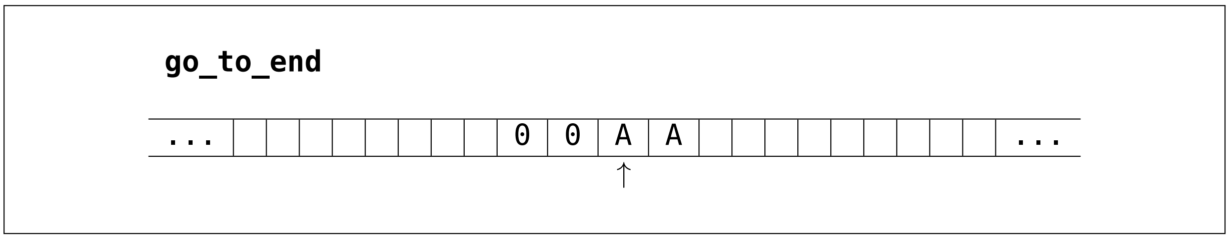


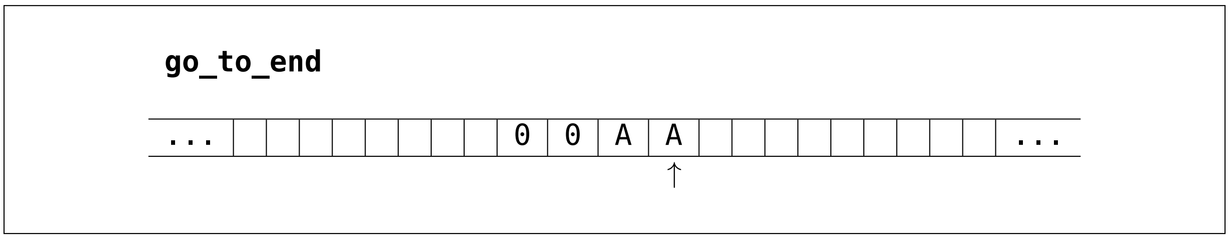

go_to_end
...
0
1
A
...

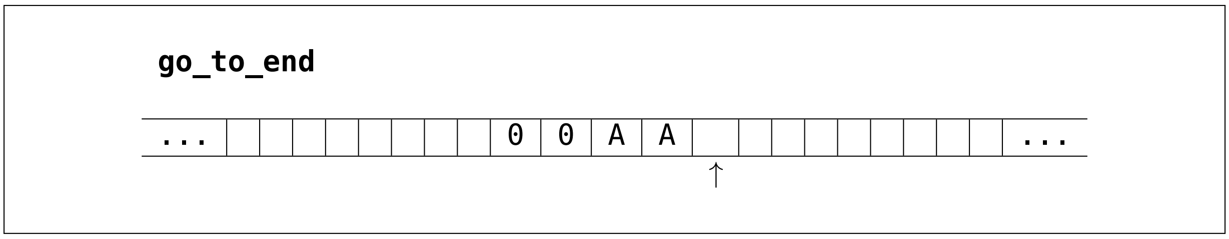

go_to_end
...
0
1
A
...

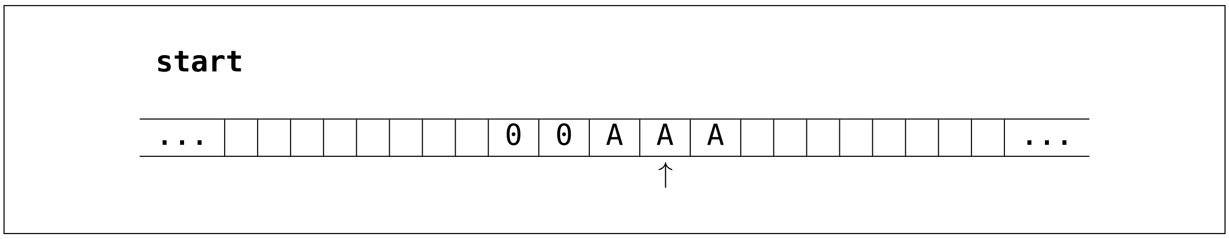

start
...
0
1
A
A
...

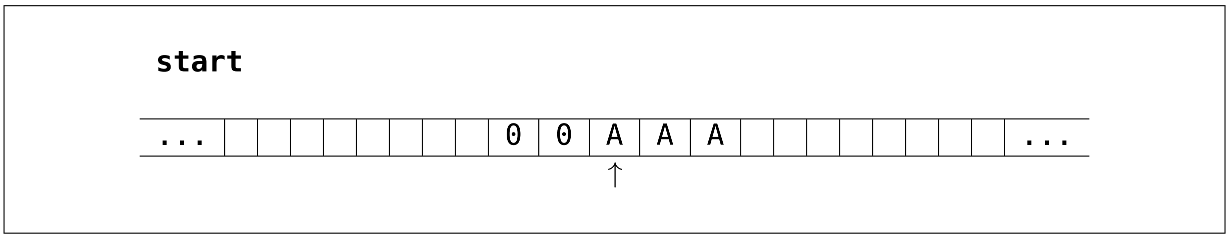

start
...
0
1
A
A
...

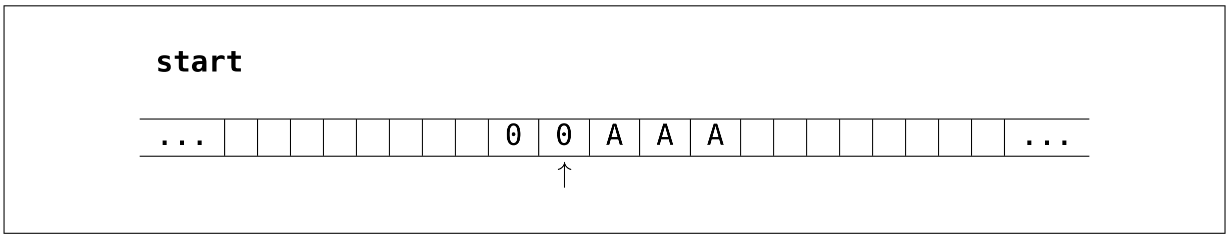

go_to_end
...
0
0
A
A
...

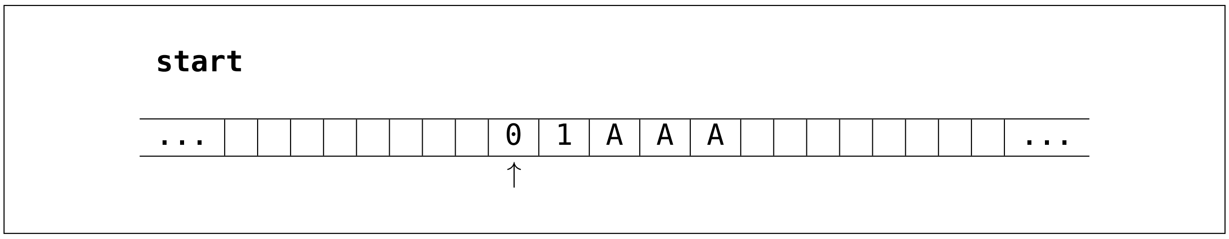

go_to_end
...
0
0
A
A
...

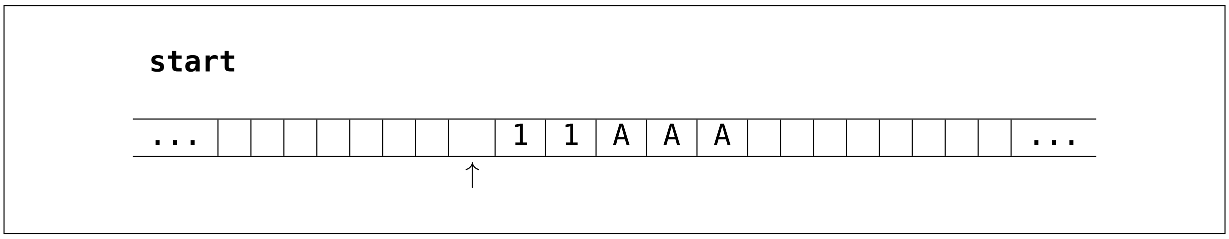

go_to_end
... 0 0 A A ...

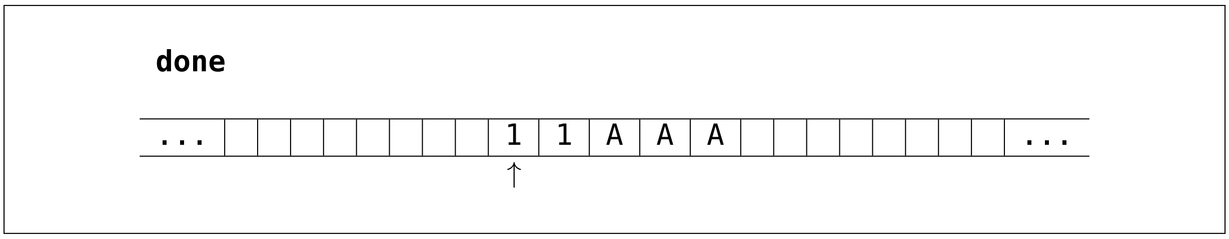

start
... 0 0 A A A ...

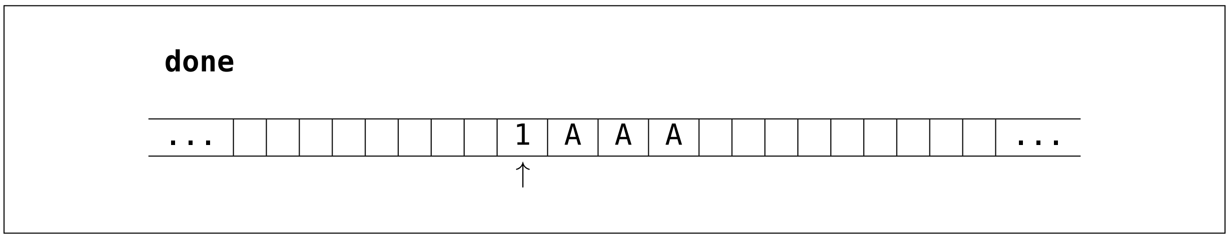

start
... 0 0 A A A ...

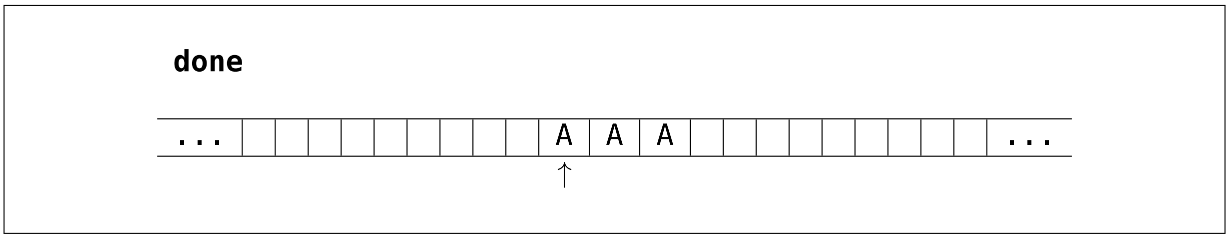

start
... 0 0 A A A ...


start
... 0 1 A A A ...


start
... 1 1 A A A ...







The machine terminates with 3 `A`s on the tape just as we expect. The exact same logic can be used to create a version of this program which works in base 10 rather than in binary.

```
Δ(start, 1) = (0, →, go_to_end)
Δ(start, 2) = (1, →, go_to_end)
Δ(start, 3) = (2, →, go_to_end)
Δ(start, 4) = (3, →, go_to_end)
Δ(start, 5) = (4, →, go_to_end)
Δ(start, 6) = (5, →, go_to_end)
Δ(start, 7) = (6, →, go_to_end)
Δ(start, 8) = (7, →, go_to_end)
Δ(start, 9) = (8, →, go_to_end)
Δ(start, 0) = (9, ←, start)
Δ(start, ) = (, →, done)
Δ(done, 9) = (, →, done)
Δ(go_to_end, 0) = (0, →, go_to_end)
Δ(go_to_end, 1) = (1, →, go_to_end)
Δ(go_to_end, 2) = (2, →, go_to_end)
Δ(go_to_end, 3) = (3, →, go_to_end)
Δ(go_to_end, 4) = (4, →, go_to_end)
Δ(go_to_end, 5) = (5, →, go_to_end)
Δ(go_to_end, 6) = (6, →, go_to_end)
Δ(go_to_end, 7) = (7, →, go_to_end)
Δ(go_to_end, 8) = (8, →, go_to_end)
```

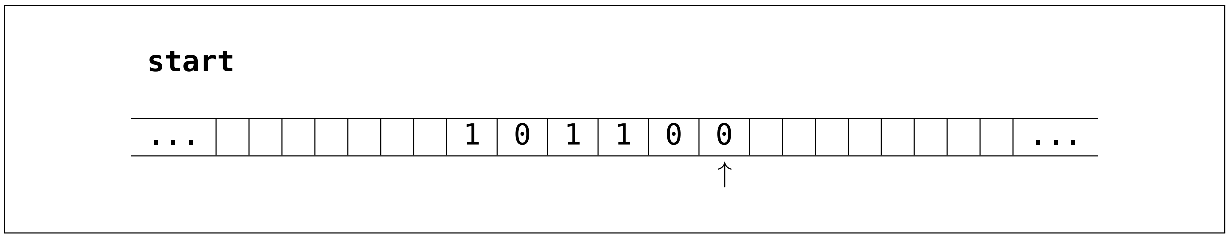
```
Δ(go_to_end, 9) = (9, →, go_to_end)
Δ(go_to_end, A) = (A, →, go_to_end)
Δ(go_to_end, ) = (A, ←, start)
Δ(start, A) = (A, ←, start)
```

2.3 Example 2: Reversing a string

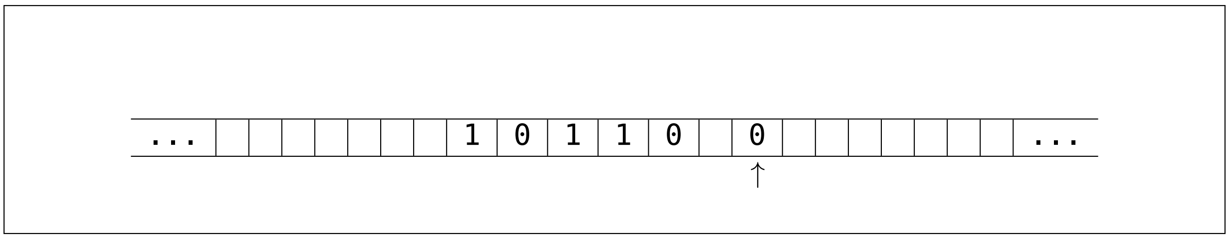
Now we can move on to a trickier example. Suppose our input is some finite sequence of non-empty symbols, and we want to write a program which reverses that sequence.

For simplicity's sake, we will limit the input sequence to be made up of `0`s and `1`s only, but the logic we will use generalises to any finite alphabet.

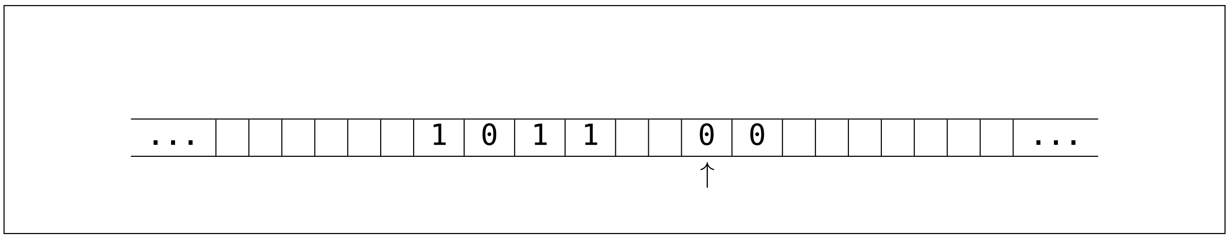
We will again assume that the machine starts in state **`start`**, and the read/write head is positioned over the rightmost cell of the input. For example, if the input was `101100`:



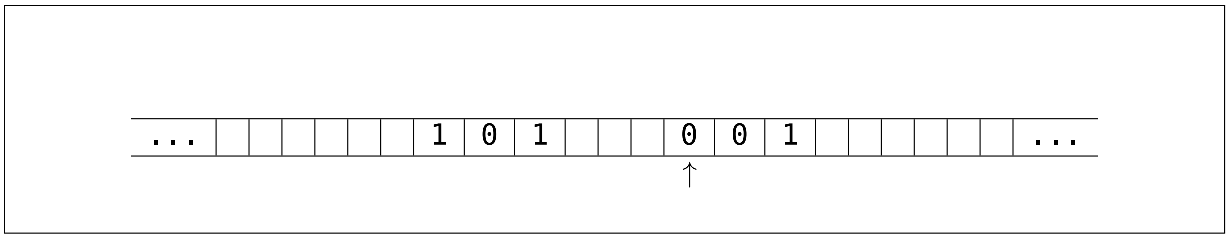
There are of course many ways to write this program, but we might suppose we want to move this symbol one cell to the right like so:



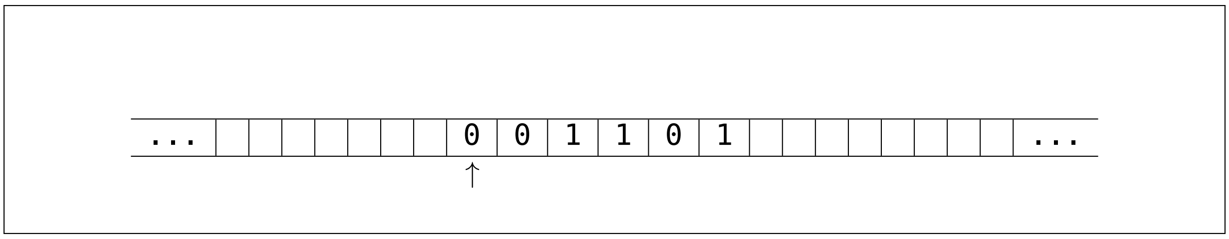
Now we have divided the sequence into two sections. The section on the left is the part of the input which we still need to reverse, and the part on the right is the part we've already reversed. We might hope that we are able to move the next digit over to the right in a similar way, all the way to the end of the right-most section.



And again,

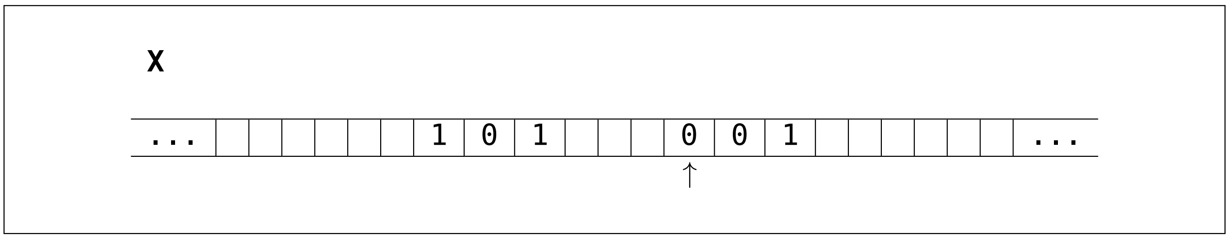


and so on, until eventually we reach a state like this:

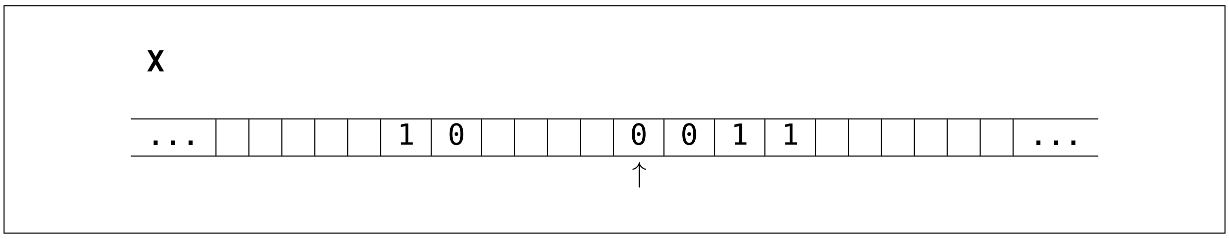


At first this seems like a good outline, because it looks like the same operation being carried out over and over again, which means that we can use a finite number of states. However, the main issue with this approach is that there is no way to know when the process has finished.

Let's take a closer look at this strategy. Ignoring the start of the program for the moment, suppose we are at one of the intermediate steps, for example:



We are in some state **X**, and we want some series of rules which we can apply to get us to the next intermediate step:

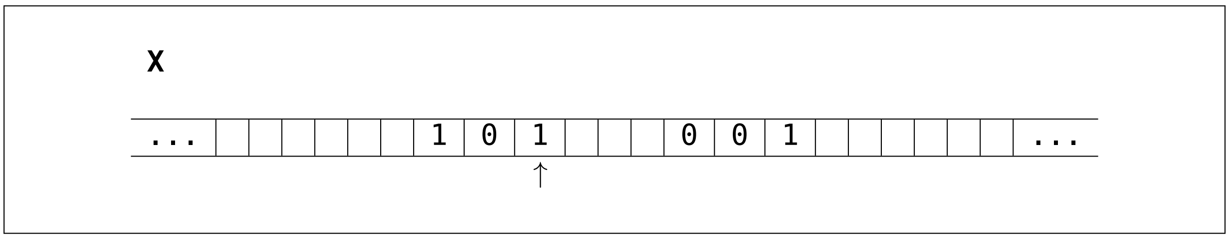


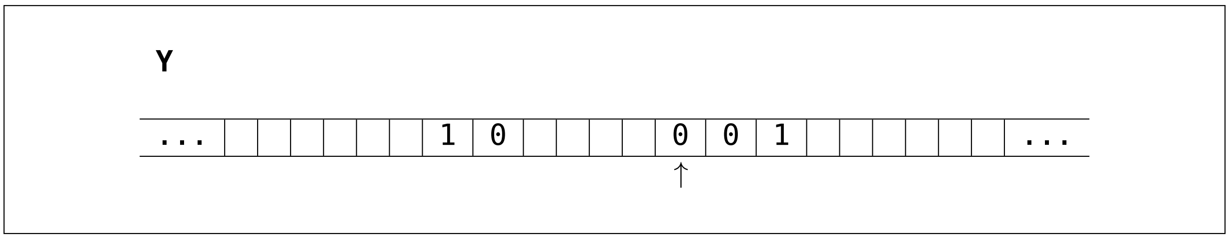
at which point we have come back to state **X**.

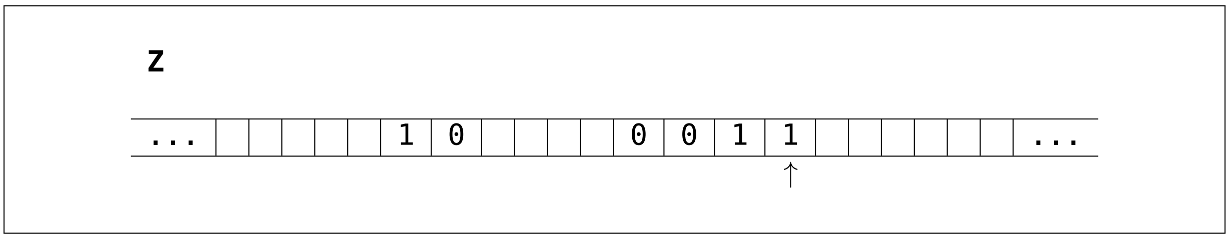
In plain English, we can describe this series of rules as carrying out the following procedure (we don't yet have to worry about how exactly we would write Turing machine rules to carry it out):

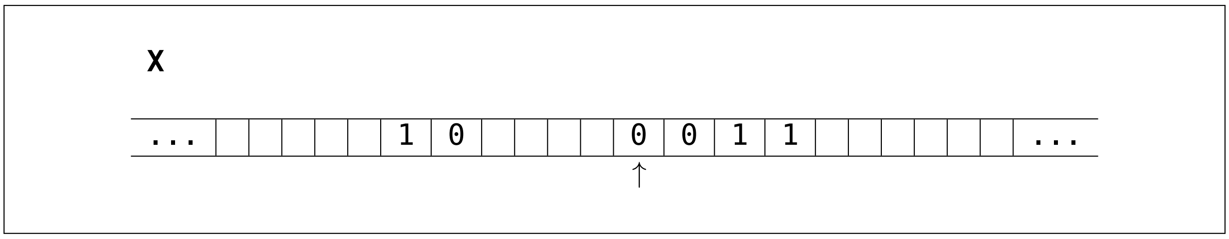
1. Move leftwards past all of the empty cells until you reach a `1` or a `0`.
2. Whichever symbol you see, replace it with an empty cell and then start moving rightwards over all of the empty cells until you reach a `1` or a `0`.
3. Continue moving to the right until you reach an empty cell. Replace it with whichever symbol you found in step 1. (Remembering which symbol to drop here is a very important topic which we will cover later in the chapter using a construct called "meta-states".)
4. Go back to the leftmost cell of the right-hand section.

These four steps are shown below for our example intermediate steps:



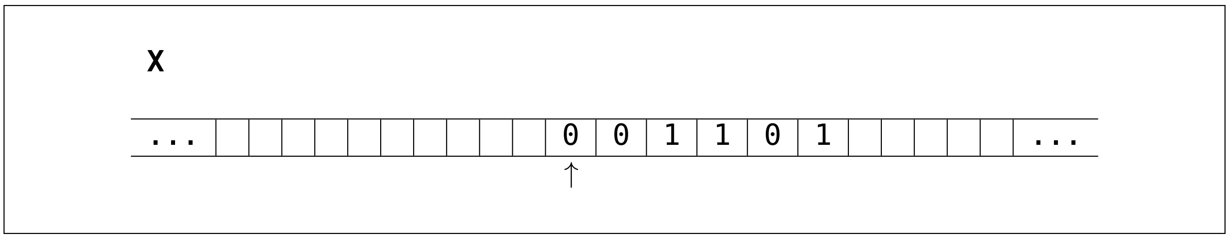






In order to be able to use a finite number of rules, it would be convenient if we implemented this procedure in such a way that it can now repeat again to get us to the next intermediate step, and then to the next, and so on.

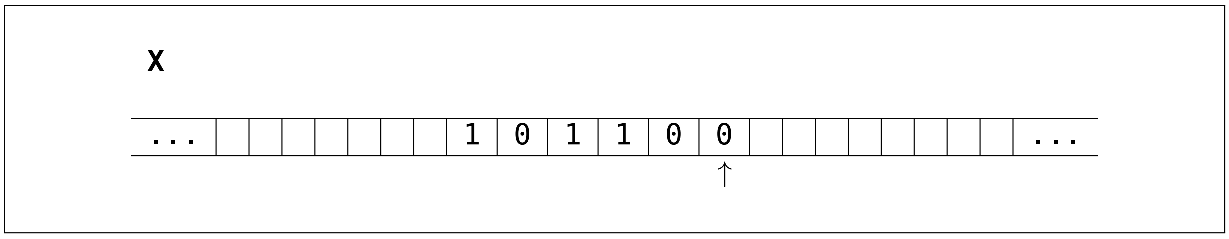
However, at each intermediate step, the size of the gap between the two sections increases by one cell. For arbitrarily large inputs, this gap will in turn become arbitrarily large. This becomes an issue once every symbol is copied over from the left section to the right.

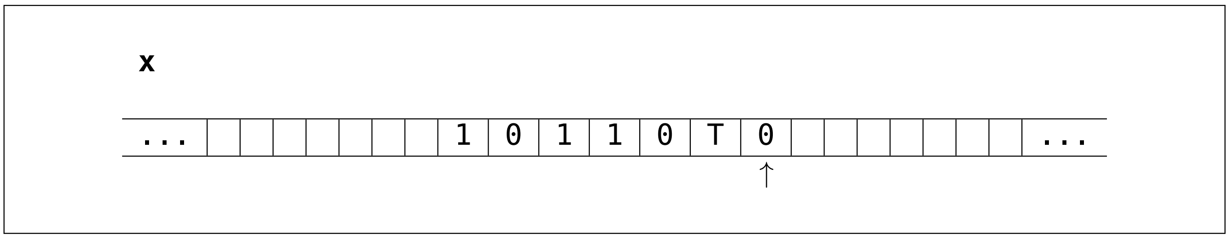


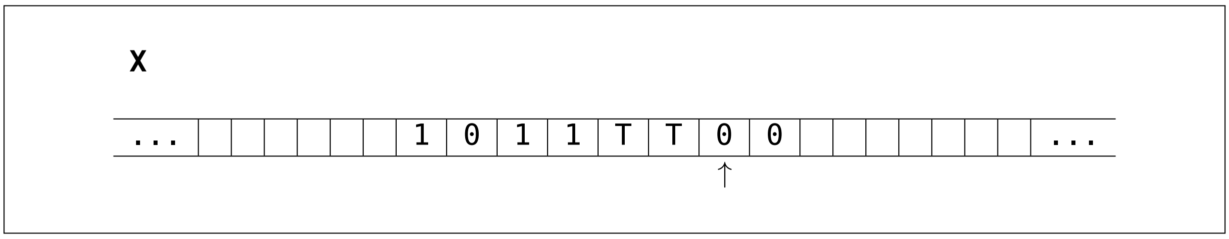
Sure, the sequence has been successfully reversed, but we're still in state **X**. That means that step 1. is about to occur, but now there are no more digits to reach, and our doomed little read/write head will sail off into the infinite leftward void, never to return.

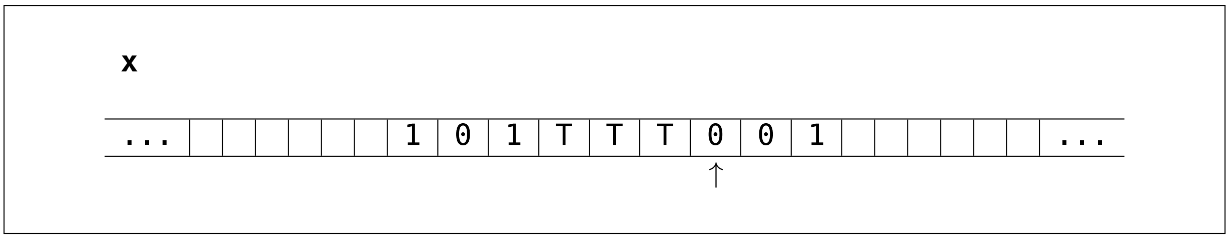
In short, this program will fail because it has no way to differentiate between the task being complete, and what is simply a very large gap between the two parts of the sequence.

We can address this problem in many ways, but in my eyes the most simple is to just expand our alphabet by one symbol, say T. This symbol can be arbitrary, but it needs to be a symbol which cannot appear in the input sequence. Now, instead of leaving an empty gap between the two sections, we can leave behind a line of T symbols. The intermediate steps might look something like this:

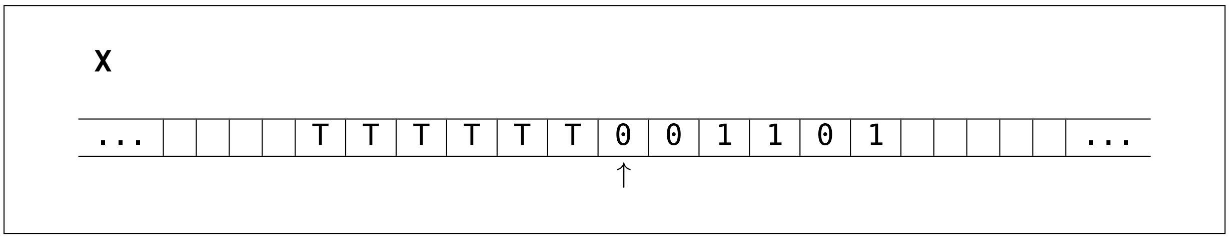






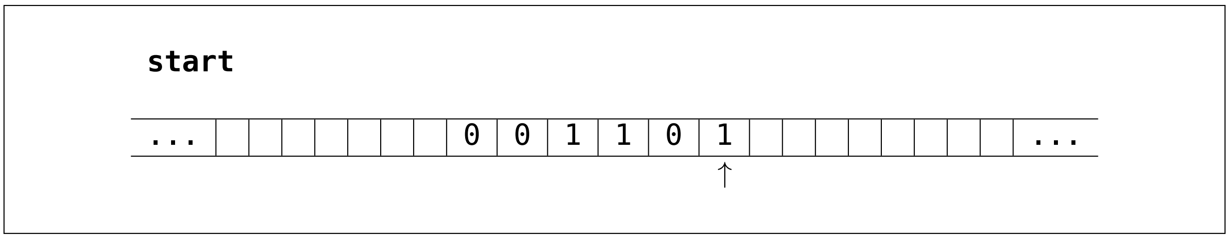


and so on, until eventually we end up with something like this:



It is now fairly clear to see how the machine can tell that the sequence has finished reversing. It will start off moving left trying to cross the gap of `T`s like normal, but instead of reaching a `1` or a `0` like for the intermediate steps, it will reach an empty cell. This will be our indicator that the job is done. We can then clean up the `T`s we dropped, and terminate gracefully.

So once again, we start in a scenario like this:

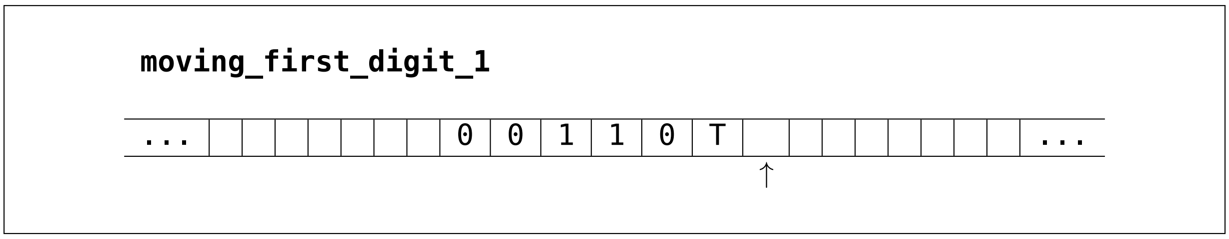


Remember, the diagram is only a visualisation aid for this particular input. We need to consider all of the possible situations for any valid input. So, if we are in the **`start`** state and we see the empty symbol, we are done. The sequence is empty, and so the reversed sequence is also empty. We move in either direction and transition to some termination state.

```
Δ(start, ) = (, →, done)
```

Otherwise, if we see a `0` or a `1`, we need to replace it with a `T` and move one cell to the right. However, we can't transition into the same state in both cases. The cell we've now moved into will always be empty, so if we are in the same state, seeing the same symbol, then we'd have to place the same symbol down. This will not work, because in the first case (when the symbol was a `0`) we want to drop a `0`, and in the second case (when the symbol was a `1`) we want to drop a `1`. Due to this limitation, we need to use two separate states. We will call them **`moving_first_digit_0`** and **`moving_first_digit_1`**.

```
Δ(start, 0) = (T, →, moving_first_digit_0)
Δ(start, 1) = (T, →, moving_first_digit_1)
```



As mentioned above, from each of these new states, we need to drop the appropriate symbol. Then we can move left, and go into the state which previously we called **X**. For the sake of readability, here we will name this state **fetch**, because it is the state we are in when we are going to fetch the next digit to carry across. Note that since we no longer need to remember any digits from earlier, we can go back to using a single state for **fetch**.

```
Δ(moving_first_digit_0, ) = (0, ←, fetch)
Δ(moving_first_digit_1, ) = (1, ←, fetch)
```

This is an incredibly important notion when writing any non-trivial programs for a Turing machine, and it is a technique I call “carrying”.

moving_first_digit_0 and **moving_first_digit_1** are both used for pretty much exactly the same purpose, but we had to use two states because the machine has to remember which one of the symbols it saw earlier. If the input sequences had been able to consist of n different symbols, then we would have had to use n different states for this.

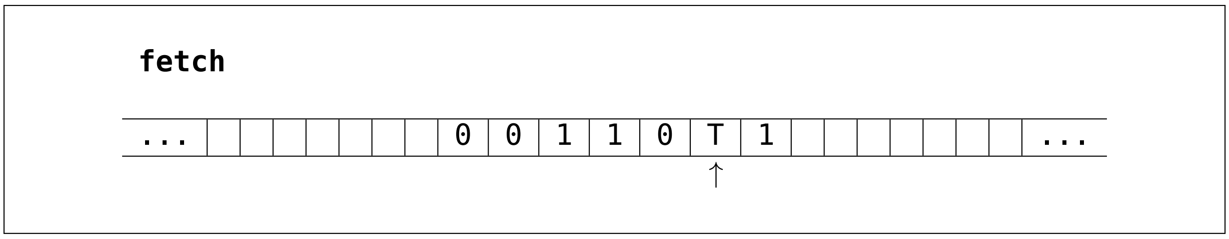
To help simplify our mental model of the state space, we will define an object called a “meta-state”. For example, **moving_first_digit_0** and **moving_first_digit_1** are states, but combined, they form the meta-state **moving_first_digit**. We say that this meta-state is carrying one symbol from the set $\{0, 1\}$. Since it is carrying one symbol, we say it is a first degree meta-state.

If hypothetically you wanted the machine to be in some state **foo** but you needed it to remember one symbol from the set $\{0, 1\}$ and one symbol from the set $\{A, B, C\}$, then you would say that **foo** is a second degree meta-state, consisting of the following actual states:

foo_0_A,
foo_0_B,
foo_0_C,
foo_1_A,
foo_1_B,
foo_1_C

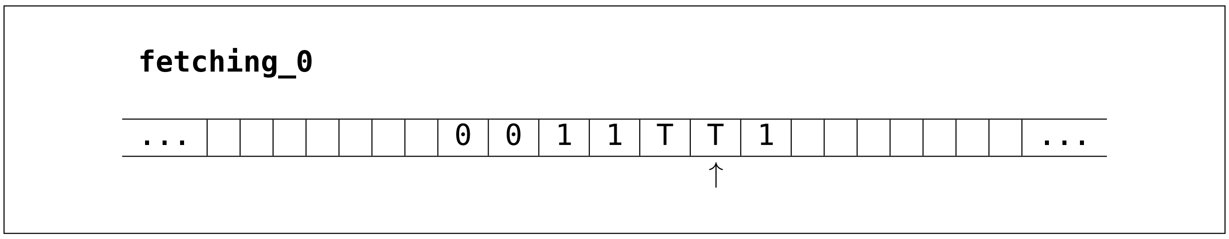
Remember that this is simply to help us think about our algorithms. The Turing machine itself cannot work with meta-states.

Going back to our example, we left off here:



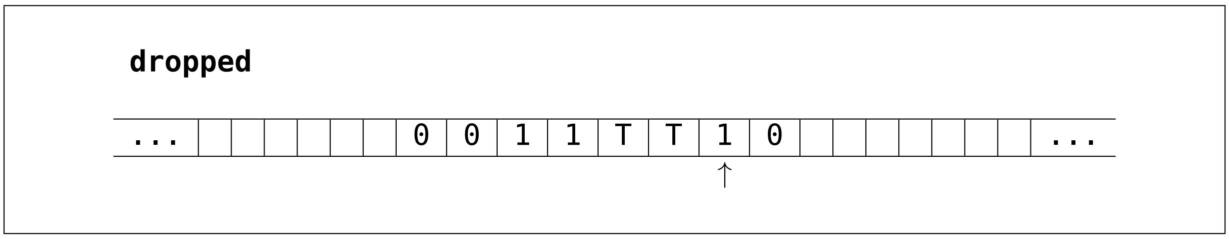
Now we need to travel left until we hit a `0` or `1`, at which point we can replace it with a `T`, move to the right, and move into meta-state **`fetching`** which carries a symbol from $\{0, 1\}$.

```
Δ(fetch, T) = (T, ←, fetch)
Δ(fetch, 0) = (T, →, fetching_0)
Δ(fetch, 1) = (T, →, fetching_1)
```



From here we travel rightwards until we reach an empty cell, at which point we drop whichever symbol we are carrying, and move back towards the left, into a state called **`dropped`**

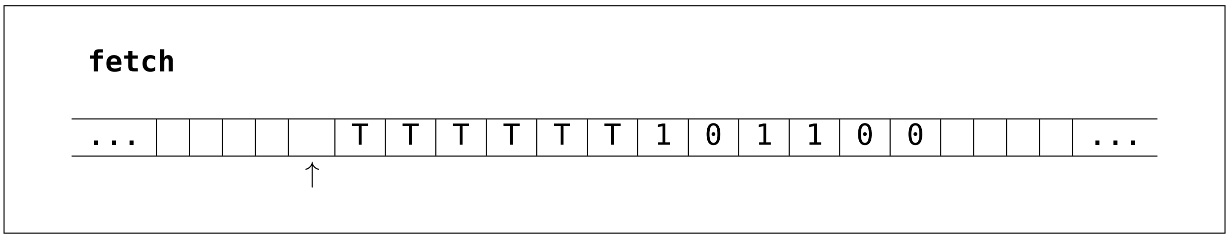
```
Δ(fetching_0, T) = (T, →, fetching_0)
Δ(fetching_1, T) = (T, →, fetching_1)

Δ(fetching_0, 0) = (0, →, fetching_0)
Δ(fetching_1, 0) = (0, →, fetching_1)

Δ(fetching_0, 1) = (1, →, fetching_0)
Δ(fetching_1, 1) = (1, →, fetching_1)

Δ(fetching_0, ) = (0, ←, dropped)
Δ(fetching_1, ) = (1, ←, dropped)
```



From here, we just need to travel leftwards until we reach the string of Ts at which point we can go back to state **fetch**.

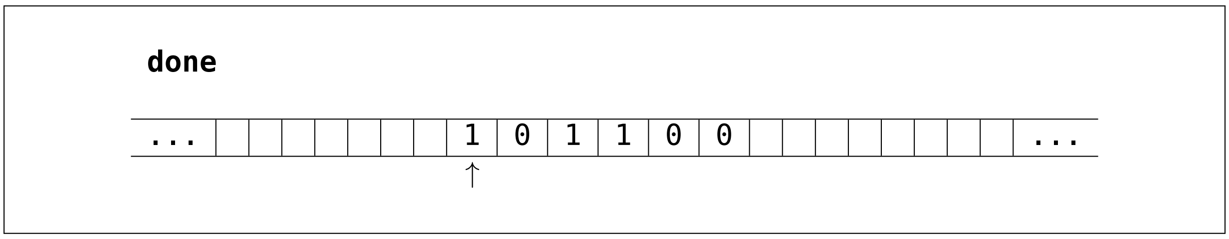
```
Δ(dropped, 0) = (0, ←, dropped)
Δ(dropped, 1) = (1, ←, dropped)
Δ(dropped, T) = (T, ←, fetch)
```

We've now created a loop, which will continue to run through all of the intermediate steps until eventually we run out of digits to **fetch**.



At this point we've finished the task, so we just need to travel back to the output string, clearing up the Ts as we go.

```
Δ(fetch, ) = (, →, done)
Δ(done, T) = (, →, done)
```



Lo and behold, the sequence is successfully reversed.

All in all, there are 21 transition rules which define this program.

```
Δ(start, ) = (, →, done)

Δ(start, 0) = (T, →, moving_first_digit_0)
Δ(start, 1) = (T, →, moving_first_digit_1)

Δ(moving_first_digit_0, ) = (0, ←, fetch)
Δ(moving_first_digit_1, ) = (1, ←, fetch)
```

```
Δ(fetch, T) = (T, ←, fetch)
Δ(fetch, 0) = (T, →, fetching_0)
Δ(fetch, 1) = (T, →, fetching_1)

Δ(fetching_0, T) = (T, →, fetching_0)
Δ(fetching_1, T) = (T, →, fetching_1)
Δ(fetching_0, 0) = (0, →, fetching_0)
Δ(fetching_1, 0) = (0, →, fetching_1)
Δ(fetching_0, 1) = (1, →, fetching_0)
Δ(fetching_1, 1) = (1, →, fetching_1)

Δ(fetching_0, ) = (0, ←, dropped)
Δ(fetching_1, ) = (1, ←, dropped)

Δ(dropped, 0) = (0, ←, dropped)
Δ(dropped, 1) = (1, ←, dropped)
Δ(dropped, T) = (T, ←, fetch)

Δ(fetch, ) = (, →, done)
Δ(done, T) = (, →, done)
```

Due to the nature of meta-states, the numbers of rules and states grow exponentially with the size of the alphabet. This will be something to bear in mind when we come to design our pseudo-assembly language.

3 Designing a Machine Code Language

3.1 Defining the Format

In the previous chapter we represented each of our transition rules in the form of Δ function notation. This, however, is rather slow and difficult for a Turing Machine simulator to parse. Instead we'll want to design some sort of efficient machine code structure which can encode our set of rules.

There are of course many ways to do this, but the one we will use is the following:

> The first 6 bytes of the machine code will be 0x545552494E47, which is the ASCII representation of the string `"TURING"`. This will make it clear to any parser what type of file it is dealing with.
>
> Next will be the unsigned integer representing how many rules there are in the set.
>
> The next part of the machine code will be the unsigned integer representing the starting state of the first rule.
>
> Then, a C-style null-terminated string representing the symbol which must be seen in order for the first rule to apply.
>
> Then, another C-style string representing the symbol which will be put into the current cell when the first rule is applied.
>
> Next comes a single byte, the least significant bit of which represents the direction in which the read/write head moves after applying the first rule. Arbitrarily we can say that a value of 0 means left, and a value of 1 means right.
>
> Then, another unsigned integer representing the ending state of the first rule.
>
> The five segments above are repeated for each rule in the ruleset.
>
> After this, optionally, is some metadata which we will discuss in more detail later.

The structure described above has some interesting properties. One of these is that we have switched to using integers to represent the state names, rather than strings. For example, we would have state **`0`**, state **`1`** etc., instead of state **`start`**, state **`done`** etc. The reason we do this is that it will allow the Turing machine simulator to more efficiently work out which rule (if any) applies at any given time.

However, this presents us with a problem. Namely, how do we encode integers of arbitrary length? In most programming languages integers are stored as values of finite length (typically 4 or 8 bytes), however this would give us only a fixed number of states to work with. Instead we can use an encoding called *variable length quantity* (VLQ) which was originally designed to encode arbitrarily large integers for MIDI files.

Suppose we have some unsigned integer x which we want to encode — for example, 2862441. We will write this number in binary:

1010111010110101101001

Next we pad the left of this bit string with 0s until the number of bits is a multiple of 7.

0000001010111010110101101001

Now we split this bit string into chunks of length 7.

0000001 0101110 1011010 1101001

Next we prepend a 1 to each of the chunks, except the last one, to which we instead prepend a 0. Now, the most significant bit of each 8-bit chunk (or octet) represents whether or not the integer has more bytes to come.

10000001 10101110 11011010 01101001

Converting this to hex, it means that our number, 2862441, can be written as the bytes 0x81AEDA69.

The fact that we are using integers instead of strings to represent our states also means that our program will be harder to write, interpret, and debug, as we would have to somehow remember which state corresponds to which concept. Strings give us a handy mnemonic device to remember what our states are for, and using them allows for mental models such as meta-states. To aid in creating the programs, in the next section we will write some C++ code which allows us to specify our program with string states, and they will be automatically converted into integer states when outputting machine code.

Furthermore, to aid with debugging, this C++ tool can optionally record a mapping from state integers to state strings in the metadata of the machine code program. This will allow the Turing machine simulator to tell us the string associated with the current state at any given time. This will be covered in more detail later when we discuss the metadata.

Another quirk of this machine code format we've specified is that a whole byte is used to represent the read/write head direction for each rule, when only the least significant bit actually encodes information. This is a redundancy of 7 bits per rule. For a large ruleset, this wasted space will add up very quickly. The reason we do this is so that every data element in the machine code is aligned to an 8-bit boundary, which will allow for much cleaner, faster parsing of the ruleset by the Turing machine simulator.

Let's run through an example instruction to see how this encoding works in practice. Suppose we have the following program from page 11:

```
Δ(start, 1) = (0, →, go_to_end)
Δ(start, 0) = (1, ←, start)
Δ(start, ) = (, →, done)
Δ(done, 1) = (, →, done)
Δ(go_to_end, 0) = (0, →, go_to_end)
Δ(go_to_end, 1) = (1, →, go_to_end)
Δ(go_to_end, A) = (A, →, go_to_end)
Δ(go_to_end, ) = (A, ←, start)
Δ(start, A) = (A, ←, start)
```

We can assume that states are assigned increasing integers in order of when they are first mentioned. That is to say, **`start`** → **`0`**, **`go_to_end`** → **`1`**, and **`done`** → **`2`**.

We would expect to see the following machine code when viewed in hex:

```
54 55 52 49 4E 47 09 00 31 00 30 00 01 01 00 30 00 31 00 00 00 00
 ↪  00 00 01 02 02 31 00 00 01 02 01 30 00 30 00 01 01 01 31 00 31
 ↪  00 01 01 01 41 00 41 00 01 01 01 00 41 00 00 00 00 41 00 41 00
 ↪  00 00
```

To make sense of this, here it is again but formatted more helpfully, and with comments indicating what each section represents:

```
54 55 52 49 4E 47       // ASCII string "TURING"

09                      // VLQ encoding of the number 9 — the
 ↪  number of rules.

                        // 1st rule
00                      // VLQ encoding of 0 — the "start" state
31 00                   // C-style string "1"
30 00                   // C-style string "0"
```

```
01                      // Move read/write head to the right
01                      // VLQ encoding of 1 — the "go_to_end"
 ↪  state

                        // 2nd rule
00                      // State 0 ("start")
30 00                   // "0"
31 00                   // "1"
00                      // Left
00                      // State 0 ("start")

                        // 3rd rule
00                      // State 0 ("start")
00                      // Empty C-style string "" — the empty
 ↪  symbol
00                      // Empty C-style string "" — the empty
 ↪  symbol
01                      // Right
02                      // State 2 ("done")

                        // 4th rule
02                      // State 2 ("done")
31 00                   // "1"
00                      // ""
01                      // Right
02                      // State 2 ("done")

                        // 5th rule
01                      // State 1 ("go_to_end")
30 00                   // "0"
30 00                   // "0"
01                      // Right
01                      // State 1 ("go_to_end")

                        // 6th rule
01                      // State 1 ("go_to_end")
31 00                   // "1"
31 00                   // "1"
01                      // Right
01                      // State 1 ("go_to_end")

                        // 7th rule
01                      // State 1 ("go_to_end")
41 00                   // "A"
41 00                   // "A"
01                      // Right
01                      // State 1 ("go_to_end")

                        // 8th rule
01                      // State 1 ("go_to_end")
00                      // ""
```

```
41 00                       // "A"
00                          // Left
00                          // State 0 ("start")

                            // 9th rule
00                          // State 0 ("start")
41 00                       // "A"
41 00                       // "A"
00                          // Left
00                          // State 0 ("start")
```

As you can imagine, writing this machine code manually would be a very time consuming and error-prone process. That is why in the next section we will write an encoder which will generate this for us.

3.2 Creating an Encoder

Finally, it's time to write some code! As previously mentioned, this we will be using C++14 to create a tool which allows us to specify our program using string states, and will output machine code. We will create a project directory and inside of it we will have a subdirectory named `src`, and a Makefile. Inside the `src` directory we will create two files: `encoder.cc`, and `encoder.hh`. The project directory should look like:

```
[Project Directory]
├── src/
│   ├── encoder.cc
│   └── encoder.hh
└── Makefile
```

We will also want a `dist` directory in the project, but we can let the Makefile handle this. For now, our Makefile will look like this:

Makefile

```
CC = g++
CFLAGS = -std=c++14 -g -Wall

encoder: dist-dir
	$(CC) $(CFLAGS) -o dist/encoder.o -c src/encoder.cc

dist-dir:
	mkdir -p dist

clean:
	rm -rf dist
```

As you can see, we are using the g++ compiler for our encoder. The `clean` target will delete the entire `dist` directory, and the `dist-dir` target will create it if it doesn't already exist. That way we can say that the `encoder` target depends on the `dist-dir` target, and we won't get caught out trying to output to a directory that doesn't exist.

We will be adapting this Makefile as we add more components to the project.

The first thing the encoder needs is a **`struct`** which will contain the data of a given rule in the ruleset. Since this will be shared by components other than the encoder, we will factor it out into a separate header file, `src/rule.hh`.

src/rule.hh

```
#pragma once
#include <string>

typedef unsigned long long state_t;

enum Direction
{
    LEFT = 0,
    RIGHT = 1
};

struct Rule
{
    state_t startState;
    std::string matchSymbol;
    std::string replaceSymbol;
    Direction direction;
    state_t endState;
};
```

There are some key things to note here. The first is that we are using `unsigned long long` to store our state indices as that is the largest appropriate built-in datatype provided by C++. However, throughout the code we will refer to this datatype as `state_t`. Not only does this save time to write, but it will also allow us to switch this out for a different type later if we need to (for example, if we end up with so many states that we need a custom datatype to handle such a large number).

Another thing to note is that we have assigned a value of `0` to the enum value `LEFT` and a value of `1` to the enum value `RIGHT`. While this may well be done automatically by the compiler, it is worth specifying it ourselves because it documents the fact that in our machine code, the left direction is encoded by the byte 0 and the right direction is encoded by the byte 1.

Apart from these two facts, everything else should be fairly self-explanatory. Now we can go to `encoder.hh` and outline how we want our encoder to work.

src/encoder.hh

```cpp
#pragma once
#include <vector>
#include <unordered_map>
#include "rule.hh"

class Encoder
{
private:
    state_t m_stateCount;
    std::vector<Rule> m_rules;
    std::unordered_map<std::string, state_t> m_stateNames;

    state_t getStateIndex(const std::string &name);
    inline void encodeRule(std::ostream &, const Rule &) const;

public:
    Encoder(void);

    void addRule(const std::string &startState,
                 const std::string &match,
                 const std::string &replacement,
                 const Direction direction,
                 const std::string &endState);
    void output(std::ostream &);
};
```

Let’s dive into why we’ve defined the class in this way.

Other classes will use the `Encoder` class by calling its `addRule` method, passing it the details of the a rule. Vitally, this call will be using strings to represent the state names. This method will be called for each rule, and then finally the `output` method will be called, which will output the machine code for all of the added rules into some output stream. The reason we can't output the machine code of each rule as soon as we add it (and hence not have to hold them in memory) is that the number of rules has to be encoded right at the start of the output, and this number is not known until all of the rules have been added. This decision will also make things easier when we discuss hacking in an I/O system to our simulator, but that will be covered in more detail later.

In order to assign each state a number, the encoder will need to keep track of how many states have been defined so far, which is the purpose of the `m_stateCount` member.

The currently added rules will be stored in the vector `m_rules`.

The map `m_stateNames` will map state strings to state indices. Furthermore there is a method `getStateIndex` which also performs this mapping. This method, however, will also add the given state to the map if it's not in there already.

Finally, it will also be helpful to have a method `encodeRule` which will output an individual rule's machine code to some output stream. For the reasons mentioned previously, we cannot expose this method to the public to allow them to encode states as they are added. Instead this method will be called from within the `output` method as a way of factoring out some of the functionality. The fact that it will only be used here is what makes it worth **`inline`**-ing.

We can start implementing this class in `encoder.cc`. We'll start with the constructor.

src/encoder.cc

```
#include "encoder.hh"

Encoder::Encoder(void)
{
        m_stateNames["start"] = 0;
        m_stateCount = 1;
}
```

The constructor takes no arguments and simply reserves state **`start`** as 0.

The next method we can implement is `getStateIndex`. This method simply looks for the given string in the `m_stateNames` map. If it doesn't find it, then it adds it in with the next available integer, and returns that integer.

src/encoder.cc

```
state_t Encoder::getStateIndex(const std::string &name)
{
    auto search = m_stateNames.find(name);
    if (search == m_stateNames.end())
    {
        state_t index = m_stateCount++;
        m_stateNames[name] = index;
        return index;
    }
    return search->second;
}
```

Next, we will write the `addRule` method. It takes in as its input the starting state, the symbol to match, the symbol with which to replace it, the read/write head direction, and the end state of a rule. Note here that the states are being passed in as strings.

All this method needs to do is find the corresponding state indices for each of the start and end states, and then add an instance of the `Rule` struct to the vector `m_rules`.

src/encoder.cc

```
void Encoder::addRule(const std::string &startState,
                      const std::string &match,
                      const std::string &replacement,
                      const Direction direction,
                      const std::string &endState)
{

    state_t startStateIndex = getStateIndex(startState);
    state_t endStateIndex = getStateIndex(endState);

    Rule rule{startStateIndex, match, replacement, direction,
 ↪  endStateIndex};

    m_rules.push_back(rule);
}
```

Now we can move on to the actual encoding. To do this we will need some method which takes in a state index and writes the encoded VLQ octets to an output stream. Since eventually we will need to create another method which can read VLQ values from an input stream when we implement the decoder, let's factor this functionality out into a separate translation unit, `src/vlq.cc` with a corresponding header `src/vlq.hh`.

The header will be relative straightforward.

src/vlq.hh

```
#pragma once
#include "rule.hh"

namespace vlq
{
    void writeToStream(std::ostream &, state_t);
    state_t readFromStream(std::istream &);
}
```

At the moment we only need to implement `writeToStream`, and we will do so in `vlq.cc`. The first piece of logic is that if the input number is 0, we only need to output a null byte to the stream and then return, as this is the VLQ encoding for the number 0.

src/vlq.cc

```
#include <ostream>
#include <stack>
#include "vlq.hh"

void vlq::writeToStream(std::ostream &os, state_t value)
{
    if (value == 0)
    {
        os << '\0';
        return;
    }
```

Next we will create a stack of octets. Each octet will be represented by an `unsigned char` because this is also an 8-bit value. The reason we use a stack is because with the operation `value & 0x7F` we can retrieve the least significant 7 bits of the value first, but this needs to be the last octet in the VLQ. As such, we store each octet on a stack so that we can output each one as we pop it, in the reverse order to that in which we pushed them.

```
src/vlq.cc

    std::stack<unsigned char> octets;
    while (value > 0)
    {
        unsigned char o = 0x80;
        o |= value & 0x7F;
        value >>= 7;
        octets.push(o);
    }
```

Initialising the octet to `0x80` sets the most significant bit to 1, which in VLQ encoding means that there will be more octets to follow. Although we don't want to set this bit for the last octet, we don't have a way to tell whether the current octet is the last one at this point in the program, so we will simply unset the bit for the last octet before we output it.

Here we are also shifting the value to the right by 7 bits, meaning that on the next iteration of the loop we will get the subsequent 7 bits.

Finally, we iterate over the stack, popping each octet, unsetting its most significant bit if this happens to be the last octet, and then outputting it to the stream.

```
src/vlq.cc

    size_t size = octets.size();
    for (size_t i = 0; i < size; i++)
    {
        auto octet = octets.top();
        octets.pop();

        if (i == size - 1)
        {
            octet &= 0x7f;
        }

        os << octet;
    }
}
```

Since we've added new source files we must go back and update our Makefile to add a new target.

Makefile

```
vlq: dist-dir
	$(CC) $(CFLAGS) -o dist/vlq.o -c src/vlq.cc
```

The `encoder` target does not need to depend on the `vlq` target because we don't necessarily need to recompile the VLQ code every time we recompile the encoder. We just have to make sure that when we make an executable that relies on the encoder, it links with both `dist/encoder.o` *and* `dist/vlq.o`.

We can now go back to `encoder.cc` and include `vlq.hh`. We will also include `ostream` because we will now implement the method `encodeRule`.

src/encoder.cc

```
#include <ostream>
#include "encoder.hh"
#include "vlq.hh"
```

src/encoder.cc

```
inline void Encoder::encodeRule(std::ostream &os, const Rule &rule)
 ↪   const
{
    vlq::writeToStream(os, rule.startState);
    for (char c : rule.matchSymbol)
    {
        os << c;
    }
    os << '\0';
    for (char c : rule.replaceSymbol)
    {
        os << c;
    }
    os << '\0';
    os << (unsigned char)rule.direction;
    vlq::writeToStream(os, rule.endState);
}
```

This method should be fairly self-explanatory. We start by writing the VLQ-encoded value of the start state (note that by the time this method gets called, the conversion from string states to integer states will have already been done). Then, we write each character from the matching symbol, followed by a null byte. Then the same with the replacement symbol.

For the direction, since we've ensured that the enum value `LEFT` has a value of 0 and `RIGHT` has a value of 1, we can simply cast the direction enum value to an `unsigned char`, which will give us the byte to write to the output.

Finally, we write the VLQ-encoded value of the end state.

Now that we've got a method for writing a single rule to an an output stream, it will be easy to write the `output` method which writes the whole program.

src/encoder.cc

```
void Encoder::output(std::ostream &os)
{
    os << "TURING";
    vlq::writeToStream(os, m_rules.size());
    for (Rule rule : m_rules)
    {
        encodeRule(os, rule);
    }
}
```

We start by writing the 6 bytes of the string `"TURING"` to the stream, followed by the VLQ-encoded value of the total number of states. Then we write each rule. Later we will add support for metadata but for the moment this is all we need to do.

The encoder is now in a working and usable state!

3.3 Creating a Unit Testing Framework

Before moving on it's worth creating some tests for the encoder. Additionally, it will save time in the long run if we implement our own basic unit testing framework, which we can extend to include tests for all of the other components we will add.

We will create a directory `src/tests` which will include our tests and our framework. This directory will be structured as follows:

```
src/tests/
 ├── [component]-tests.cc
 ├── [component]-tests.cc
 ├── [component]-tests.cc
 └── ⋮
```

```
│
├─ tests.cc
└─ tests.hh
```

Where `[component]` is the name of a component we'd want to test, such as "`encoder`".

For now, we will only create the following files:

```
src/tests/
├─ encoder-tests.cc
├─ tests.cc
└─ tests.hh
```

We will want some data structure to keep track of how many tests have been run, and how many of them have passed. We will define this structure in `src/tests/tests.hh`.

src/tests/tests.hh

```cpp
#pragma once
#include <iostream>

namespace tests
{
    struct Record
    {
        unsigned int passed;
        unsigned int total;

    public:
        Record &operator+=(const Record &);
    }
}

std::ostream &operator<<(std::ostream &, const tests::Record &);
```

We have declared that the `Record` struct supports the operator +=, meaning that a record can be incremented by another record. This will be useful because we can separate our tests into several test suites — one for each component — which can each have its own record, meanwhile we can also keep a master record which accumulates all of the record data for each component's tests.

We have also declared that output streams support the << operator for `Record` objects. This will allow us to easily output them to stdout.

We will define both of these methods in `src/tests/tests.cc`.

```
src/tests/tests.cc
```

```cpp
#include <iostream>
#include "tests.hh"

tests::Record &tests::Record::operator+=(const tests::Record &other)
{
    this->passed += other.passed;
    this->total += other.total;
    return *this;
}

std::ostream &operator<<(std::ostream &os, const tests::Record
 ↪  &record)
{
    return os
           << "PASSED: "
           << record.passed
           << "/"
           << record.total
           << " "
           << (record.passed == record.total ? "✅" : "❌" );
}
```

We now want to go back to `tests.hh` and declare a namespace for the encoder's tests, and a method therein called `allTests`. This method will be defined in `encoder-tests.cc`.

```
src/tests/tests.hh
```

```cpp
    namespace encoder
    {
        void allTests(Record &);
    }
```

If we had multiple components with tests, we would need to declare a new namespace and a new `allTests` method for each of them. This code is simply begging to be a preprocessor macro. You might expect that we would use a macro like this, src/tests/tests.hh

```
#define DECLARE_COMPONENT_NAMESPACE(component) \
    namespace component                        \
    {                                          \
        void allTests(Record &);               \
    }
```

at which point we might use the macro like this:

src/tests/tests.hh

```
    DECLARE_COMPONENT_NAMESPACE(encoder)
```

However, it will be easier in the long run to instead create a macro like this:

src/tests/tests.hh

```
#define FOR_ALL_COMPONENTS \
    X(encoder)
```

Which we will use as follows:

src/tests/tests.hh

```
#define X(name)                  \
    namespace name               \
    {                            \
        void allTests(Record &); \
    }

    FOR_ALL_COMPONENTS

#undef X
```

This is an incredibly helpful (though admittedly rather ugly) trick. The `FOR_ALL_COMPONENTS` macro just applies some other (as yet unknown) macro `X` to each component we want to test. When we add more components, we would simply adjust the `FOR_ALL_COMPONENTS` macro to something like this:

src/tests/tests.hh

```
#define FOR_ALL_COMPONENTS   \
    X(encoder)               \
    X(decoder)               \
    X(vlq)                   \
    X(some_other_component) \
```

Then, when we want to run some code for every component, we define the macro `X` to be that code, use `FOR_ALL_COMPONENTS`, and then undefine `X`, as we have done above.

This is useful because next we want to actually run all of the `allTests` methods for each component, and we can make use of this macro to do so succinctly.

We will write this code in `tests.cc`.

src/tests/tests.cc

```
int main()
{

    tests::Record record{0, 0};

#define X(component) tests::component::allTests(record);
    FOR_ALL_COMPONENTS
#undef X

    std::cout
        << record
        << std::endl;
    return 0;
}
```

Note how in this situation we are using the macro `X` to be one that calls the `allTests` method for a given component.

At the moment we are using a single record to store the results from all of the component tests combined. Later we will change this to use a separate record for each component, and a single master record to accumulate them.

We won't actually write any of the unit tests until we've finished developing the framework, but it would be helpful to have a placeholder one to let us test out our system. It has not escaped my notice that testing a testing framework is perhaps a rather tedious-sounding notion, but alas we must soldier on and take solace in the knowledge that we've taken the time to do things the right way.

We will put a placeholder unit test in `src/tests/encoder-tests.cc`

src/tests/encoder-tests.cc

```
#include "tests.hh"

static bool placeholderTest(void)
{
        return true;
}

void tests::encoder::allTests(tests::Record &record)
{
        record.total++;
        if (placeholderTest()) {
                record.passed++;
        }

        return;
}
```

As you can see, all that this test does is return a value of `true`. (Hooray! we passed.)

The `allTests` method increments the number of recorded tests, and if the placeholder test passes, it also increments the recorded number of passed tests.

To build the tester we must add a target to the Makefile.

Makefile

```
tests: dist-dir
        $(CC) $(FLAGS) -o dist/tests src/tests/encoder-tests.cc \
                                     src/tests/tests.cc
```

We can now build and run our tester with the following commands:

```
make tests
./dist/tests
```

To which we are pleasantly greeted with the output:

```
PASSED: 1/1 ✅
```

If we were to change our unit test to return `false` instead of `true`, we would instead see:

```
PASSED: 0/1 ❌
```

As you can see, this would not be a particularly helpful output if we had more than one test. We would want to see exactly which tests failed, and for which components. To achieve this, we will create a method `tests::perform` which takes in a name for the test, a function reference to the test, and a record reference to modify. We will declare this method in `tests.hh`.

First we must include `functional`,

```
src/tests/tests.hh

#include <functional>
```

and then we can declare the method.

```
src/tests/tests.hh

    void perform(const std::string &,
                 const std::function<bool()> &,
                 Record &);
```

We will define this method in `tests.cc`.

```
src/tests/tests.cc

void tests::perform(const std::string &name,
                    const std::function<bool()> &testToPerform,
                    tests::Record &record)
{
    std::cout << "   " << name << ": ";
    bool passed = testToPerform();
    std::cout << (passed ? "✅" : "❌") << std::endl;
    record.passed += passed;
    record.total++;
}
```

We can now go back to `encoder-tests.cc` and modify it to use this method.

src/tests/encoder-tests.cc

```cpp
void tests::encoder::allTests(tests::Record &record)
{
    tests::perform("placeholderTest", placeholderTest, record);

    return;
}
```

While this is much simpler, there is still some redundancy in the fact that we have to write the test name twice, once as a string to use as the test name, and once again to get the function reference. We can solve this minor inconvenience using another preprocessor macro, which does this for us. Since all of the components' tests will want to use this macro, we will put it in `tests.hh`.

src/tests/tests.hh

```cpp
#define RUN_TEST(testFunc, record) \
    tests::perform(#testFunc, (testFunc), (record))
```

Here we are using `#testFunc` to get the macro parameter `testFunc` as a string, saving us the need to write it out twice. We can now simplify `allTests`.

src/tests/encoder-tests.cc

```cpp
void tests::encoder::allTests(tests::Record &record)
{
    RUN_TEST(placeholderTest, record);

    return;
}
```

If we rebuild and run our tester, we will see the following output:

```
 placeholderTest: ✅
PASSED: 1/1 ✅
```

This is more helpful as it tells us exactly which tests passed and which tests failed, as well as a total.

The final feature to add to our tester is some output displaying which components the tests are for, and how many of the tests passed for that component. Similarly to the `tests::perform` method, we can create a new wrapper method called `tests::performComponentTests` which accepts as parameters a string to use as the name of the component, a function reference to that component's `allTests` method, and a record reference to update.

We will declare it in `tests.hh`.

src/tests/tests.hh

```
    void performComponentTests(const std::string &,
                               const std::function<void(Record &)>
↪   &,
                               Record &);
```

We will then define it in `tests.cc`.

src/tests/tests.cc

```
void tests::performComponentTests(const std::string &name,
                                  const
↪   std::function<void(tests::Record &)> &testsToPerform,
                                  tests::Record &record)
{
    std::cout << "Running " << name << " tests:" << std::endl;
    tests::Record innerRecord{0, 0};
    testsToPerform(innerRecord);
    std::cout << innerRecord << std::endl
              << std::endl;
    record += innerRecord;
}
```

Notice how here we are constructing a new instance of `Record` just for this individual component, which we are outputting once all of that component's tests have been run. Then, we increment the master record by this new one (which is why we needed to implement `tests::Record::operator+=`).

We will modify our main method to use this wrapper.

src/tests/tests.cc

```cpp
int main()
{

    tests::Record record{0, 0};

#define X(component)                                               \
    tests::performComponentTests(#component,                       \
                                 tests::component::allTests,  \
                                 record);
    FOR_ALL_COMPONENTS
#undef X

    std::cout
        << "OVERALL "
        << record
        << std::endl;
    return 0;
}
```

After rebuilding and running the tester, we will see the following output:

```
Running encoder tests:
  placeholderTest: ✅
PASSED: 1/1 ✅

OVERALL PASSED: 1/1 ✅
```

With this, the unit testing framework is complete. While it may seem over-engineered and bulky, it is so versatile and extendable that it will make a world of difference when it comes to writing a great number of tests across several different components.

3.4 Unit Tests for the Encoder

After all that work, it would be great to see just how easy unit tests are to write in our framework.

We will call our first test `simpleTest` and it will use our example from page 29,

```
Δ(start, 1) = (0, →, go_to_end)
Δ(start, 0) = (1, ←, start)
Δ(start, ) = (, →, done)
```

```
Δ(done, 1) = (, →, done)
Δ(go_to_end, 0) = (0, →, go_to_end)
Δ(go_to_end, 1) = (1, →, go_to_end)
Δ(go_to_end, A) = (A, →, go_to_end)
Δ(go_to_end, ) = (A, ←, start)
Δ(start, A) = (A, ←, start)
```

for which we would expect the following machine code:

```
54 55 52 49 4E 47 09 00 31 00 30 00 01 01 00 30 00 31 00 00 00 00
 ↪  00 00 01 02 02 31 00 00 01 02 01 30 00 30 00 01 01 01 31 00 31
 ↪  00 01 01 01 41 00 41 00 01 01 01 00 41 00 00 00 00 41 00 41 00
 ↪  00 00
```

To perform this test we will need a helper function which can convert a string into hex.

We will put this in `src/tests/encoder-tests.cc`. We will need to include the `iomanip` and `sstream` libraries. While we're at it, let's include `encoder.hh` because we'll need it soon.

src/tests/encoder.cc

```cpp
#include <sstream>
#include <iomanip>
#include "tests.hh"
#include "../encoder.hh"

std::string stringToHex(const std::string &input)
{
    std::stringstream hex;
    for (const unsigned char c : input)
    {
        hex
            << std::uppercase
            << std::setfill('0')
            << std::setw(2)
            << std::hex << (int)c;
    }
    return hex.str();
}
```

This function works by iterating over every character from the string and converting it to hex, padding it with a `'0'` on the left if it's only one digit, and setting it to uppercase (just so we don't have to worry about type-insensitive string comparisons).

We can now write our simple unit test.

src/tests/encoder-tests.cc

```
static bool simpleTest(void)
{
    Encoder encoder;
    encoder.addRule("start", "1", "0", Direction::RIGHT,
    "go_to_end");
    encoder.addRule("start", "0", "1", Direction::LEFT, "start");
    encoder.addRule("start", "", "", Direction::RIGHT, "done");
    encoder.addRule("done", "1", "", Direction::RIGHT, "done");
    encoder.addRule("go_to_end", "0", "0", Direction::RIGHT,
    "go_to_end");
    encoder.addRule("go_to_end", "1", "1", Direction::RIGHT,
    "go_to_end");
    encoder.addRule("go_to_end", "A", "A", Direction::RIGHT,
    "go_to_end");
    encoder.addRule("go_to_end", "", "A", Direction::LEFT, "start");
    encoder.addRule("start", "A", "A", Direction::LEFT, "start");

    std::ostringstream os;

    encoder.output(os);

    auto result = stringToHex(os.str());

    std::string expected =
        "545552494E4709003100300001010030"
        "00310000000000000102023100000102"
        "01300030000101013100310001010141"
        "00410001010100410000000041004100"
        "0000";

    return result == expected;
}
```

We also need to modify our `allTests` method to run this test.

src/tests/encoder-tests.cc

```
void tests::encoder::allTests(tests::Record &record)
{
    RUN_TEST(simpleTest, record);

    return;
}
```

For now, this is the only test we will write for the encoder itself. However, we will also test the VLQ encoding method we implemented. We will put this in its own file, `src/tests/vlq-tests.cc`.

src/tests/vlq-tests.cc

```
#include <sstream>
#include "tests.hh"
#include "../vlq.hh"

void tests::vlq::allTests(tests::Record &record) {
    return;
}
```

To make sure that this component's tests are run by the tester, all we have to do is modify the `FOR_ALL_COMPONENTS` macro to also use this component.

src/tests/tests.hh

```
#define FOR_ALL_COMPONENTS \
    X(encoder)             \
    X(vlq)
```

For our first test we can check that the number 0 is correctly encoded as a null byte.

src/tests/vlq-tests.cc

```
static bool is0EncodedCorrectly(void)
{
    std::stringstream os;
    vlq::writeToStream(os, 0);
    return os.str() == std::string("\0", 1);
}
```

You may notice that for the comparison we use the expression `std::string("\0", 1)`. We need to specify the length of the string in the constructor because otherwise it assumes that the first argument is a C-style null-terminated string, and so will construct the empty string instead of a string containing a null byte.

For our next test we can ensure that a number which should only occupy a single octet is encoded correctly. Let's use 50. The expected octet is 0x32.

src/tests/vlq-tests.cc

```
static bool is50EncodedCorrectly(void)
{
    std::stringstream os;
    vlq::writeToStream(os, 50);
    return os.str() == "\x32";
}
```

Finally we'll add a test for the multi-octet case. Let's use 123456789. The expected octets are 0xBAEF9A15.

src/tests/vlq-tests.cc

```
static bool is123456789EncodedCorrectly(void)
{
    std::stringstream os;
    vlq::writeToStream(os, 123456789);
    return os.str() == "\xBA\xEF\x9A\x15";
}
```

Now we can modify `tests::vlq::allTests` to actually run these tests.

src/tests/vlq-tests.cc

```
void tests::vlq::allTests(tests::Record &record)
{

    RUN_TEST(is0EncodedCorrectly, record);
    RUN_TEST(is50EncodedCorrectly, record);
    RUN_TEST(is123456789EncodedCorrectly, record);

    return;
}
```

All we need to do now is modify the `tests` target in the Makefile to also compile `src/tests/vlq-tests.cc`.

```
Makefile

tests: vlq encoder
        $(CC) $(FLAGS) -o dist/tests dist/encoder.o \
                                     dist/vlq.o \
                                     src/tests/vlq-tests.cc \
                                     src/tests/encoder-tests.cc \
                                     src/tests/tests.cc
```

Now when we compile and run the tester we will be thrilled to see a lot of green ticks.

```
Running encoder tests:
  simpleTest: ✅
PASSED: 1/1 ✅

Running vlq tests:
  is0EncodedCorrectly: ✅
  is50EncodedCorrectly: ✅
  is123456789EncodedCorrectly: ✅
PASSED: 3/3 ✅

OVERALL PASSED: 4/4 ✅
```

3.5 Creating a Decoder

Now that we've got a working encoder which lets us transform our ruleset into machine code, we can start creating a decoder which allows us to turn machine code back into a ruleset.

One method which we will of course need is `vlq::readFromStream` which we delcared in a previous section but have not yet implemented. This method is supposed to read a VLQ-encoded value from an input stream and return it as a `state_t`.

We will define this method in `src/vlq.cc`.

src/vlq.cc

```cpp
state_t vlq::readFromStream(std::istream &is)
{
    state_t value = 0;
    bool moreOctets;
    do
    {
        unsigned char octet = is.get();
        moreOctets = (octet & 0x80) >> 7;
        value <<= 7;
        value |= octet & 0x7F;
    } while (moreOctets);
    return value;
}
```

It starts off with a value of zero, and then reads an octet from the input stream. It appends the least significant 7 bits of this octet to the running total, and repeats until it finds an octet whose most significant bit is a 0, indicating that the current octet is the last one for this value.

We will also need a data structure to contain our decoded Turing machine program once we decode it. For the moment, this could simply be a `std::vector<Rule>` object, but when we add program metadata to the machine code format, this structure will need to hold some additional information. For now, we will describe a simple wrapper struct in a new header, `src/program.hh`.

src/program.hh

```cpp
#pragma once
#include <vector>
#include "rule.hh"

struct Program
{
    std::vector<Rule> rules;
};
```

Using these tools, we will be able to implement the decoder. We will do so in a new file, `src/decoder.cc`, with a corresponding header file `src/decoder.hh`.

Like for the encoder, the decoder logic will be contained within a `Decoder` class. However, since we no longer need to worry (for the moment) about translating between string states and integer states, this class will be much simpler than `Encoder`. It will have a single method, `decode`, which takes in an input stream and returns a pointer to one of our `Program` objects.

src/decoder.hh

```
#pragma once
#include <memory>
#include <istream>
#include "program.hh"

class Decoder
{
public:
    std::unique_ptr<Program> decode(std::istream &);
};
```

The reason this method returns a `Program` pointer rather than simply a `Program` is so that we can return a **`nullptr`** if the decoding fails.

We can define this method in `decoder.cc`. The first thing we want to do is check that the 6 magic bytes `"TURING"` appear as the first 6 bytes of the stream.

src/decoder.cc

```
#include "decoder.hh"
#include "vlq.hh"

std::unique_ptr<Program> Decoder::decode(std::istream &is)
{
    std::string magicBytes(6, '\0');
    is.readsome(&magicBytes[0], 6);

    if (magicBytes != "TURING")
    {
        return nullptr;
    }
```

Next we want to initialise a `Program` object to contain our decoded data. We can use `vlq::readFromStream` to read the value encoding the number of rules in the ruleset. We'll use this value to set the size of the program's `rules` vector.

src/decoder.cc

```
    Program program;

    state_t numRules = vlq::readFromStream(is);
    program.rules.resize(numRules);
```

Now, we can iterate over each rule and read its data from the input stream, storing it in the appropriate `Rule` object in the `rules` vector.

src/decoder.cc

```
    for (state_t i = 0; i < numRules; i++)
    {
        program.rules[i].startState = vlq::readFromStream(is);

        std::getline(is, program.rules[i].matchSymbol, '\0');
        std::getline(is, program.rules[i].replaceSymbol, '\0');

        program.rules[i].direction =
 ↪  static_cast<Direction>(is.get());

        program.rules[i].endState = vlq::readFromStream(is);
    }
```

Finally, we return a pointer to the `Program` object we've created.

src/decoder.cc

```
    return std::make_unique<Program>(program);
}
```

Note that the only error-checking that we are doing is for the magic bytes. We are not checking that we have actually managed to decode the expected number of rules, whether `vlq::readFromStream` fails due to reaching the end of the stream without finding a terminating octet, or indeed any other way in which this decoder could fail with bad input.

However, this is going to be sufficient for our purposes. The magic byte check will tell us whether we have, for example, passed the decoder a file which isn't even a Turing machine code file. Otherwise, we will assume that any machine code file that we pass to the decoder will be correctly formed.

If you feel so inclined, I encourage you to modify this decoder to be more informative in the case of an error, perhaps even displaying some human-readable error messages.

To be able to compile the decoder, we must add a target to our Makefile.

Makefile

```
decoder: dist-dir
	$(CC) $(CFLAGS) -o dist/decoder.o -c src/decoder.cc
```

3.6 Unit Tests for the Decoder

We will now write some unit tests for the decoding logic we've just implemented.

Let's start with `vlq::readFromStream`. We will use the same test cases we used for `vlq::writeToStream`. Namely, 0, 50, and 123456789.

These tests are quite straight forward, and we will write them in `src/tests/vlq-tests.cc`.

src/tests/vlq-tests.cc

```
static bool is0DecodedCorrectly(void)
{
    std::stringstream is(std::string("\0", 1));
    auto result = vlq::readFromStream(is);
    return result == 0;
}

static bool is50DecodedCorrectly(void)
{
    std::stringstream is("\x32");
    auto result = vlq::readFromStream(is);
    return result == 50;
}

static bool is123456789DecodedCorrectly(void)
{
    std::stringstream is("\xBA\xEF\x9A\x15");
    auto result = vlq::readFromStream(is);
    return result == 123456789;
}
```

For decoding, we also want to test one more case. If the stream contains more than just the VLQ we want to decode, we want to make sure that the VLQ decoder stops where it is supposed to, and ignores all further bytes. For this we can use the following descriptive yet clunkily-named test:

src/tests/vlq-tests.cc

```cpp
static bool is123456789DecodedCorrectlyWithMoreBytes(void)
{
    std::stringstream is("\xBA\xEF\x9A\x15\41\41\41\41");
    auto result = vlq::readFromStream(is);
    return result == 123456789;
}
```

This test is the same as `is123456789DecodedCorrectly` but where the input string has several extra bytes at the end, which will hopefully be ignored.

We must now modify the `allTests` method to run all of these new tests.

src/tests/vlq-tests.cc

```cpp
void tests::vlq::allTests(tests::Record &record)
{

    RUN_TEST(is0EncodedCorrectly, record);
    RUN_TEST(is50EncodedCorrectly, record);
    RUN_TEST(is123456789EncodedCorrectly, record);
    RUN_TEST(is0DecodedCorrectly, record);
    RUN_TEST(is50DecodedCorrectly, record);
    RUN_TEST(is123456789DecodedCorrectly, record);
    RUN_TEST(is123456789DecodedCorrectlyWithMoreBytes, record);

    return;
}
```

Now we must test the decoder itself. We will create a new file called `src/tests/decoder-tests.cc`.

src/tests/decoder-tests.cc

```cpp
#include <sstream>
#include "tests.hh"
#include "../decoder.hh"

void tests::decoder::allTests(Record &record)
{
    return;
}
```

We will create a unit test named `simpleTest` which is similar to that of the encoder. It too will use the example from page 29, passing in the machine code of the program and then checking whether the decoded rules match what we'd expect.

src/tests/decoder-tests.cc

```cpp
static bool simpleTest()
{
    std::stringstream is(std::string(
    "\x54\x55\x52\x49\x4E\x47\x09\x00\x31\x00\x30\x00\x01\x01\x00\x30"
    "\x00\x31\x00\x00\x00\x00\x00\x00\x01\x02\x02\x31\x00\x00\x01\x02"
    "\x01\x30\x00\x30\x00\x01\x01\x01\x31\x00\x31\x00\x01\x01\x01\x41"
    "\x00\x41\x00\x01\x01\x01\x00\x41\x00\x00\x00\x00\x41\x00\x41\x00"
        "\x00\x00", 66));

    Decoder decoder;
    auto program = decoder.decode(is);
    if (!program)
        return false;
    if (program->rules.size() != 9)
        return false;
    if (!(program->rules[0] == Rule{0, "1", "0", Direction::RIGHT, 1}))
        return false;
    if (!(program->rules[1] == Rule{0, "0", "1", Direction::LEFT, 0}))
        return false;
    if (!(program->rules[2] == Rule{0, "", "", Direction::RIGHT, 2}))
        return false;
    if (!(program->rules[3] == Rule{2, "1", "", Direction::RIGHT, 2}))
        return false;
    if (!(program->rules[4] == Rule{1, "0", "0", Direction::RIGHT, 1}))
        return false;
    if (!(program->rules[5] == Rule{1, "1", "1", Direction::RIGHT, 1}))
        return false;
    if (!(program->rules[6] == Rule{1, "A", "A", Direction::RIGHT, 1}))
        return false;
    if (!(program->rules[7] == Rule{1, "", "A", Direction::LEFT, 0}))
        return false;
    return program->rules[8] == Rule{0, "A", "A", Direction::LEFT, 0};
}
```

There is one piece missing from this test. We are comparing each rule to the expected output using the == operator, which we have not implemented on the `Rule` struct. Luckily, this is not too difficult to do, and we will do so in `src/rule.hh`.

src/rule.hh

```
    inline bool operator==(const Rule &o)
    {
        if (startState != o.startState)
            return false;
        if (matchSymbol != o.matchSymbol)
            return false;
        if (replaceSymbol != o.replaceSymbol)
            return false;
        if (direction != o.direction)
            return false;
        return endState == o.endState;
    }
```

Now we can go back to `decoder-tests.cc` and modify its `allTests` method to run this test.

src/tests/decoder-tests.cc

```
void tests::decoder::allTests(Record &record)
{
    RUN_TEST(simpleTest, record);
    return;
}
```

Next, we must extend the `FOR_ALL_COMPONENTS` macro in `src/tests/tests.hh` to include the `decoder` component.

src/tests/tests.hh

```
#define FOR_ALL_COMPONENTS \
    X(encoder)             \
    X(decoder)             \
    X(vlq)
```

Finally, we can edit the `tests` target in the Makefile to depend on the `decoder` target, and to also link with both the `decoder.o` object file, and the `decoder-tests.cc` source file.

```
Makefile

tests: vlq encoder decoder
	$(CC) $(FLAGS) -o dist/tests dist/encoder.o \
	                             dist/decoder.o \
	                             dist/vlq.o \
	                             src/tests/vlq-tests.cc \
	                             src/tests/encoder-tests.cc \
	                             src/tests/decoder-tests.cc \
	                             src/tests/tests.cc
```

Now if we build and run our tests, we will see the following glorious output:

```
Running encoder tests:
  simpleTest: ✅
PASSED: 1/1 ✅

Running decoder tests:
  simpleTest: ✅
PASSED: 1/1 ✅

Running vlq tests:
  is0EncodedCorrectly: ✅
  is50EncodedCorrectly: ✅
  is123456789EncodedCorrectly: ✅
  is0DecodedCorrectly: ✅
  is50DecodedCorrectly: ✅
  is123456789DecodedCorrectly: ✅
  is123456789DecodedCorrectlyWithMoreBytes: ✅
PASSED: 7/7 ✅

OVERALL PASSED: 9/9 ✅
```

What we now have is a method of encoding our rules as a binary file, and then reading in the binary file to reconstruct our rules. We will eventually use the former to create a tool which compiles a pseudo-assembly language into Turing machine code, but first we will use the latter to create a Turing machine simulator.

Our encoder and decoder are far from finished, and we will come back to extend them as and when we need to, but they are at a point now where they are functional enough for us to move on to the next steps of the project.

4 Creating a Turing Machine Simulator — TMVM

4.1 Implementing the Simulator

The next step in our project is creating a tool which can simulate a Turing machine, and thereby execute our machine code programs. We will call this tool *TMVM*, or "Turing Machine Virtual Machine".

We will be writing the code for TMVM in the same project as the encoder and decoder. At this point, our project directory should contain the following files and subdirectories:

```
[Project Directory]
├─src/
│ ├─tests/
│ │ ├─decoder-tests.cc
│ │ ├─encoder-tests.cc
│ │ ├─tests.cc
│ │ ├─tests.hh
│ │ └─vlq-tests.hh
│ ├─decoder.cc
│ ├─decoder.hh
│ ├─encoder.cc
│ ├─encoder.hh
│ ├─program.hh
│ ├─rule.hh
│ ├─vlq.cc
│ └─vlq.hh
└─Makefile
```

To create our simulator, we must have some sort of data structure which can encapsulate the tape. We will do so in a new file, `src/tape.hh`, in which we will first define a struct to be used for each cell of the tape.

src/tape.hh

```cpp
#pragma once

struct Cell
{
    Cell *left{nullptr};
    Cell *right{nullptr};
    std::string symbol{""};
};
```

This implementation of `Cell` is a doubly-linked list. This is to say that each cell contains a pointer to the cell to its left and to its right. This may be a **`nullptr`** if we have not yet allocated any cells to the left or right of this cell.

Now we can create a `Tape` class which will contain a pointer to the current cell being looked at by the read/write head of the simulated Turing machine, as well as pointers to the leftmost and rightmost cells which have been allocated. This class also needs to have methods to move the read/write head to the left or the right, allocating new cells when it needs to do so. However, these methods will be private for reasons which will become clear soon.

src/tape.hh

```cpp
class Tape
{
private:
    Cell *m_current;
    Cell *m_leftmost;
    Cell *m_rightmost;

    void moveLeft(void);
    void moveRight(void);
```

The tape will be constructed with a vector of symbols to put on the tape. We will also need to define a destructor for this class, because the tape itself will be responsible for deallocating the memory used by each of its cells.

Furthermore, this class will expose a public method, `move`, which accepts as its parameter a value from the `Direction` enum. This can be an inline method which, depending on the value of the argument, will call one of the `moveLeft` or `moveRight` private methods.

To declare each of these, we need to bring in the following headers:

src/tape.hh

```
#include <vector>
#include "rule.hh"
```

src/tape.hh

```
public:
    Tape(std::vector<std::string> initialSymbols);
    ~Tape();

    inline void move(const Direction direction)
    {
        switch (direction)
        {
        case LEFT:
            return moveLeft();
        case RIGHT:
            return moveRight();
        }
        return;
    }
```

This class will also have two more methods which allow getting and setting of the symbol in the current cell.

src/tape.hh

```
    inline std::string getSymbol(void) const { return
↪   m_current->symbol; }
    inline void setSymbol(const std::string &s) const
    {
        m_current->symbol = s;
    }
```

Yet another method on this class will return a vector containing the symbols of all of the allocated cells.

src/tape.hh

```cpp
    inline std::vector<std::string> getAllSymbols(void) const
    {
        std::vector<std::string> symbols;
        const Cell *current = m_leftmost;
        while (current)
        {
            symbols.push_back(current->symbol);
            current = current->right;
        }
        return symbols;
    }
};
```

The first method we will implement is the constructor, in `src/tape.cc`.

src/tape.cc

```cpp
#include "tape.hh"

Tape::Tape(std::vector<std::string> initialSymbols)
{
    m_current = new Cell;
    m_leftmost = m_current;
    m_rightmost = m_current;
    for (auto symbol : initialSymbols)
    {
        m_current->symbol = symbol;
        moveRight();
    }
    moveLeft();
}
```

The constructor starts by allocating a single empty cell, and updating the `m_current`, `m_leftmost`, and `m_rightmost` pointers to point to that cell. Then, we iterate over every symbol in the argument, setting the current cell's symbol to each one in turn, and then moving one cell to the right. We can assume that the `moveRight` method which we will implement soon will allocate a new cell to the right of the current one, and update the `m_current` pointer to this new cell (as well as the `m_rightmost` pointer).

Finally, we move one cell to the left. This will result in the machine starting with the read/write head over the rightmost cell in the input string, rather than an empty cell to the right of this. This is arbitrary, but consistent with the convention we established in section 2.2.

Moving on to the destructor, we simply have to start at the cell pointed to by `m_leftmost`, and iterate rightwards through the linked list, deleting each cell as we encounter it.

src/tape.cc

```
Tape::~Tape()
{
    while (m_leftmost)
    {
        Cell *next = m_leftmost->right;
        delete m_leftmost;
        m_leftmost = next;
    }
}
```

Now we can implement the `moveLeft` method. We will first check whether there is no allocated cell to the left of the current one. If this is the case, we create a new cell and insert it into the doubly-linked list. Either way, we then set `m_current` to point to the cell on the left of the current one.

src/tape.cc

```
void Tape::moveLeft(void)
{
    if (!(m_current->left))
    {
        m_current->left = new Cell;
        m_current->left->right = m_current;
        m_leftmost = m_current->left;
    }
    m_current = m_current->left;
```

One minor memory optimisation we can add at this point is to check whether the rightmost allocated cell is empty. If this is the case, we can deallocate it and remove it from the linked list. That way we will not be left with a string of empty cells on the right-hand side of the tape.

src/tape.cc

```
    if (m_rightmost->symbol.length() == 0)
    {
        Cell *next = m_rightmost->left;
        next->right = nullptr;
        delete m_rightmost;
        m_rightmost = next;
    }
}
```

The implementation for `moveRight` is almost identical, except for the switching of left and right.

src/tape.cc

```
void Tape::moveRight(void)
{
    if (!(m_current->right))
    {
        m_current->right = new Cell;
        m_current->right->left = m_current;
        m_rightmost = m_current->right;
    }
    m_current = m_current->right;
    if (m_leftmost->symbol.length() == 0)
    {
        Cell *next = m_leftmost->right;
        next->left = nullptr;
        delete m_leftmost;
        m_leftmost = next;
    }
}
```

Now that we have a `Tape` class and a `Program` struct, we can create a `Simulator` class which will execute a given program on a given tape.

We will declare this class in a new header, `src/simulator.hh`, and define it in an accompanying source file, `src/simulator.cc`.

The class will have a pointer to the program it is executing, and a reference to the tape on which it is executing the program. It will also store its current state.

src/simulator.hh

```cpp
#pragma once
#include <memory>
#include "program.hh"
#include "tape.hh"

class Simulator
{
private:
    std::unique_ptr<Program> m_program;
    Tape &m_tape;
    state_t m_state{0};
```

Note that we are initialising `m_state` to 0. This corresponds to the fact that in the encoder, we reserved state 0 as the **`start`** state.

We will also need an efficient data structure which allows us to determine which rule, if any, applies at a given point in time. Since our states are stored as indices, we can use an array for this. Stored at each index of the array will be pointers to all of the rules which could apply when the current state is equal to that index. One might imagine using an array of vectors to this effect.

src/simulator.hh

```cpp
    std::vector<const Rule*> *m_ruleMap{nullptr};
```

However, we can improve further. Each of these vectors would contain pointers to rules which share a starting state, but have different matching symbols. As such, more appropriate than a vector would perhaps be a hash table, mapping the matching symbols to rule pointers, like so:

src/simulator.hh

```cpp
#include <unordered_map>
```

src/simulator.hh

```cpp
    std::unordered_map<std::string, const Rule*>
↪   *m_ruleMap{nullptr};
```

This is the data structure we will use.

As well as the constructor, we must also implement a destructor to dispose of the `m_ruleMap` array. The class will also expose a getter for the current state, and a method which simply performs one step of the simulation (i.e., applies at most a single rule).

src/simulator.hh

```cpp
public:
    Simulator(std::unique_ptr<Program>, Tape &);
    ~Simulator();

    inline state_t getState(void) const { return m_state; };
    bool step(void);
};
```

The only issue is that we don't know the required length of the `m_ruleMap` array. We need this length to be equal to the number of states so that we can use the state number as an index into this array, but we don't know the state count without looping over the entire vector of rules. Instead, we will modify the decoder to keep track of the state count for us, and store it in the `Program` struct.

src/program.hh

```cpp
struct Program
{
    std::vector<Rule> rules;
    state_t numStates{0};
};
```

After adding this field to the `Program` struct, we only need to add a small amount of code to `src/decoder.cc`.

src/decoder.cc

```cpp
#include <algorithm>
```

src/decoder.cc

```
        program.numStates = std::max({program.numStates,
                                      program.rules[i].startState,
                                      program.rules[i].endState});
    }

    program.numStates++;
```

Note that we must increment the `program.numState` member after the loop because indices are 0-indexed, but counts are 1-indexed. This is to say, if the greatest state is state `34`, then the number of states is 35.

We will also modify our decoder unit test, `simpleTest`, to check for the correct state count.

src/tests/decoder-tests.cc

```
    if (program->numStates != 3)
        return false;
```

Now, in `src/simulator.cc`, we can define the constructor of the `Simulator` class.

src/simulator.cc

```
#include "simulator.hh"

Simulator::Simulator(std::unique_ptr<Program> program, Tape &tape)
    : m_program(std::move(program)), m_tape(tape)
{
    m_ruleMap =
        new std::unordered_map<std::string,
                               const Rule *>[m_program->numStates];

    for (const Rule &rule : m_program->rules)
    {
        m_ruleMap[rule.startState][rule.matchSymbol] = &rule;
    }
};
```

First, we allocate the `m_ruleMap` array to have as many entries as there are states in the program. Then, we iterate over each rule, placing it in the appropriate spot in the array and the relevant hash table.

The destructor is quite simple.

src/simulator.cc

```cpp
Simulator::~Simulator()
{
    delete[] m_ruleMap;
}
```

Next, we can implement the core functionality of the simulator. This, of course, is the `step` method. First, we need to get the hash table containing rules which apply in the current state.

src/simulator.cc

```cpp
bool Simulator::step(void)
{
    auto rulesForThisState = m_ruleMap[m_state];
```

Then, we look up the current symbol (that is, the symbol in the current cell) in this hash table. If we find nothing, then we can return `false`, as no rule applies to the current situation. This also means that the program has finished executing.

src/simulator.cc

```cpp
    auto search = rulesForThisState.find(m_tape.getSymbol());
    if (search == rulesForThisState.end())
    {
        return false;
    }
```

On the other hand, if the search is successful, we can get the pointer to the rule which applies, and apply it.

src/simulator.cc

```cpp
    const Rule *rule = search->second;
    m_tape.setSymbol(rule->replaceSymbol);
    m_tape.move(rule->direction);
    m_state = rule->endState;
    return true;
}
```

We can now add two more targets to the Makefile.

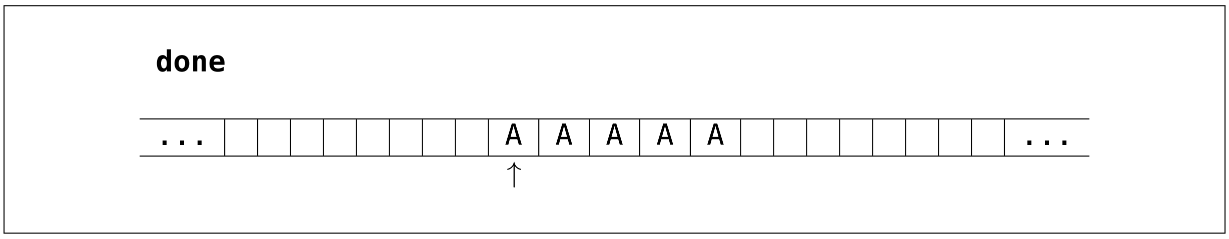
Makefile

```
simulator: dist-dir
	$(CC) $(CFLAGS) -o dist/simulator.o -c src/simulator.cc

tape: dist-dir
	$(CC) $(CFLAGS) -o dist/tape.o -c src/tape.cc
```

4.2 Outputting the Simulator

Eventually, we'll want some way to output the state of the machine and contents of the tape. That way, we'll be able to know that our program did what it was supposed to do.

First, we'll focus on outputting the contents of the tape. For example, if our tape looked like this:



Then we might want some sort of output to stdout like this:

```
| A | A | A | A | A |
  ↑
```

This output consists of 3 lines, which will cause an ugly mess if they become long and wrap. Therefore we will limit the output to only 16 cells, after which the next 16 cells will be printed underneath, and so on. This can, of course, be modified to fit your own terminal's width.

Printing the contents of the tape will involve implementing a method `std::ostream &operator<<( std::ostream &, const Tape &)`, and this method will need access to the `Tape` class's private members. Therefore we should declare this method as a `friend` of the `Tape` class.

src/tape.hh

```
#include <ostream>
```

src/tape.hh

```
    friend std::ostream &operator<<(std::ostream &, const Tape &);
```

Now, we will create a struct inside `src/tape.cc` called `TapeOutputLine`, which will encode a set of these 3 rows to be printed. This struct will also have a method which clears these three streams.

src/tape.cc

```
#include <sstream>
#define OUTPUT_CELLS_PER_ROW 16
```

src/tape.cc

```
struct TapeOutputLine
{
    std::stringstream top;
    std::stringstream middle;
    std::stringstream bottom;

    void empty(void)
    {
        top.str(std::string());
        middle.str(std::string());
        bottom.str(std::string());
    }
}
```

We want to be able to print an instance of this struct to an output stream.

```
src/tape.cc

std::ostream &operator<<(std::ostream &os, const TapeOutputLine &line)
{
    return os
           << line.top.str()
           << std::endl
           << line.middle.str()
           << std::endl
           << line.bottom.str()
}
```

Now, when we want to print the entire tape to an output stream, we can simply iterate over all of the cells in the tape and add them to an instance of this struct. Once we reach 16 cells, we print the struct followed by a newline, clear the struct, and continue.

```
src/tape.cc

std::ostream &operator<<(std::ostream &os, const Tape &tape)
{
    const Cell *current = tape.m_leftmost;
    int index = 0;

    TapeOutputLine line;
    while (current)
    {
        if (index++ >= OUTPUT_CELLS_PER_ROW)
        {
            index = 1;
            os << line << std::endl;
            line.empty();
        }

        uint len = 3 + current->symbol.length();

        line.top << std::string(len, '_');
        line.middle << "| " << current->symbol << ' ';

        for (uint i = 0; i < len; i++)
            line.bottom << "¯";

        current = current->right;
    }
    return os << line;
}
```

This code is quite ugly, but this is to be expected when trying to output what is essentially ASCII-art. Luckily, this code has no particularly interesting logic to it, and so understanding it fully is not vital. In its current state, this would output something like the following:

```
| A | A | A | A | A
```

There are 3 things missing from this output. The first is that the rightmost cell seems to be incomplete. This is because the “barrier” between each cell is added on the left-hand side before the cell’s content, and so the last cell has no barrier on the right-hand side. We can fix this by adding a barrier when we output each row.

```
src/tape.cc
```

```cpp
std::ostream &operator<<(std::ostream &os, const TapeOutputLine
    &line)
{
    return os
            << line.top.str()
            << '_'
            << std::endl
            << line.middle.str()
            << '|'
            << std::endl
            << line.bottom.str()
            << "‾";
}
```

The output would now look like this:

```
| A | A | A | A | A |
```

The next thing that’s missing is an indicator of which cell is currently under the read/write head. We can check which cell this is by simply comparing the currently printing cell to the cell `tape.m_current`.

```
src/tape.cc
```

```cpp
        bool isActive = current == tape.m_current;
```

Notice how we are using a **for** loop to add overline characters to the `bottom` string stream. We can split this loop into two, and then optionally add an arrow in between them if this cell is currently under the read/write head.

```
src/tape.cc
```

```
        for (uint i = 0; i < len / 2; i++)
            line.bottom << "¯";

        line.bottom << (isActive ? "↑" : "¯");

        for (uint i = len / 2 + 1; i < len; i++)
            line.bottom << "¯";
```

Now the output would like like this:

```
_____________________
| A | A | A | A | A |
¯¯¯↑¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯
```

The final thing that's missing is that the cell that's currently under the read/write head should be highlighted in a different colour. This can be fairly easily added in by giving our `TapeOutputLine` struct another method, `setHighlighted` which will change the colour of the output. We will also give it a flag, `isHighlighted`. Furthermore, we will edit the `empty` method to set the text to be not highlighted.

src/tape.cc

```cpp
struct TapeOutputLine
{
    std::stringstream top;
    std::stringstream middle;
    std::stringstream bottom;

    bool isHighlighted{false};

    void empty(void)
    {
        top.str(std::string());
        middle.str(std::string());
        bottom.str(std::string());
        setHighlighted(false);
    }

    void setHighlighted(bool isHighlighted)
    {
        if (isHighlighted == this->isHighlighted)
            return;
        this->isHighlighted = isHighlighted;
        std::string colorChangeChars = isHighlighted ? "\033[31m" : "\033[39m";
        top << colorChangeChars;
        middle << colorChangeChars;
        bottom << colorChangeChars;
    }
};
```

Note how here we're outputting the characters `"\033[31m"` to change the output colour to red, and `"\033[39m"` to change it back to its default colour.

Now we can add a line to the tape output method to call `setHighlighted`.

src/tape.cc

```cpp
        line.setHighlighted(isActive);
```

This is now very nearly what we want, as the output would look like this:

```
| A | A | A | A | A |
  ↑
```

As you can see, the right-hand barrier of the active cell is not being highlighted. We can fix that by changing our single call to `setHighlighted` into two calls, one to set and one to unset. Between these calls, we will draw the barrier. That means that the left-hand barrier for the cell on the immediate right of the active cell will be drawn before the highlight is unset.

src/tape.cc

```
        if (isActive)
        {
            line.setHighlighted(true);
        }

        line.top << '_';
        line.middle << '|';
        line.bottom << "‾";

        if (!isActive)
        {
            line.setHighlighted(false);
        }

        uint len = 2 + current->symbol.length();

        line.top << std::string(len, '_');
        line.middle << ' ' << current->symbol << ' ';
```

We must also edit the method which outputs the `TapeOutputLine` object to make it reset to the default colour with every new row.

src/tape.cc

```
std::ostream &operator<<(std::ostream &os, const TapeOutputLine &line)
{
    return os
           << line.top.str()
           << '_'
           << "\033[39m"
           << std::endl
           << line.middle.str()
           << '|'
           << std::endl
           << "\033[39m"
           << line.bottom.str()
           << "‾"
           << "\033[39m";
}
```

The output would now be exactly what we want:

```
_____________________
| A | A | A | A | A |
‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾
  ↑
```

Again, don't worry if this code is hard to understand. It's very difficult to write code like this cleanly, and the underlying concepts are not very interesting. It will, however, be incredibly helpful to have this code in place when it comes to testing and debugging.

Now that we can output the tape, we can use this to output the simulator. This will consist of outputting the state, and then the tape.

Like with the tape, we must declare this output method a **`friend`** of the `Simulator` class in `src/simulator.hh`

src/simulator.hh

```
#include <ostream>
```

src/simulator.hh

```
    friend std::ostream &operator<<(std::ostream &, const Simulator &);
```

Luckily, it will be much easier to implement this method than it was for the tape. We will do so in `src/simulator.cc`.

src/simulator.cc

```
std::ostream &operator<<(std::ostream &os, const Simulator &sim)
{
    os << "State: " << sim.m_state << std::endl;
    return os << sim.m_tape;
}
```

The output for the simulator would look like this:

```
State: 2
------------------------
| A | A | A | A | A |
------------------------
  ↑
```

Note that since the simulator works with integer states, this is what will be output. We will update this later to use the human-readable string states.

4.3 Testing the Simulator

We will write some simple tests for our simulator using the program from page 29, for which the machine code is:

```
54 55 52 49 4E 47 09 00 31 00 30 00 01 01 00 30 00 31 00 00 00 00
 ↪  00 00 01 02 02 31 00 00 01 02 01 30 00 30 00 01 01 01 31 00 31
 ↪  00 01 01 01 41 00 41 00 01 01 01 00 41 00 00 00 00 41 00 41 00
 ↪  00 00
```

As a reminder, this program takes a multi-cell input in binary, and outputs a string of `"A"`s which is that many characters long.

We will put these tests in `src/tests/simulator-tests.cc`. The first test will be with an empty input. We would expect the simulator to finish in state **`2`** (**`done`**) and the tape should be empty.

src/tests/simulator-tests.cc

```cpp
#include <sstream>
#include "tests.hh"
#include "../decoder.hh"
#include "../simulator.hh"

static bool emptyStringOfAsTest(void)
{
    std::stringstream is(std::string(
        "\x54\x55\x52\x49\x4E\x47\x09\x00\x31\x00\x30\x00\x01\x01\x00\x30"
        "\x00\x31\x00\x00\x00\x00\x00\x00\x01\x02\x02\x31\x00\x00\x01\x02"
        "\x01\x30\x00\x30\x00\x01\x01\x01\x31\x00\x31\x00\x01\x01\x01\x41"
        "\x00\x41\x00\x01\x01\x01\x00\x41\x00\x00\x00\x00\x41\x00\x41\x00"
        "\x00\x00",
        66));
    Decoder decoder;
    auto program = decoder.decode(is);

    std::vector<std::string> initialSymbols;
    Tape tape(initialSymbols);

    Simulator sim(std::move(program), tape);

    while (sim.step())
    {
    }

    std::vector<std::string> expected({""});

    return sim.getState() == 2 && tape.getAllSymbols() == expected;
}
```

As you can see, we start by loading the machine code into a string stream, and then decode it using the decoder to get a pointer to the program. Then, we create an empty tape (as the input is empty for this test), and then use these to construct the simulator.

We then run the `sim.step` method until it returns `false`, which indicates that the program has finished running. Then we check that the ending state is indeed **2** and that the tape ends up with a single empty cell.

The next test will be very similar except it will test what happens when the input is a single cell containing the symbol `"0"`. The expected output is the same.

```
src/tests/simulator-tests.cc

static bool stringOf0AsTest(void)
{
    std::stringstream is(std::string(
    "\x54\x55\x52\x49\x4E\x47\x09\x00\x31\x00\x30\x00\x01\x01\x00\x30"
    "\x00\x31\x00\x00\x00\x00\x00\x00\x01\x02\x02\x31\x00\x00\x01\x02"
    "\x01\x30\x00\x30\x00\x01\x01\x01\x31\x00\x31\x00\x01\x01\x01\x41"
    "\x00\x41\x00\x01\x01\x01\x00\x41\x00\x00\x00\x00\x41\x00\x41\x00"
        "\x00\x00",
        66));
    Decoder decoder;
    auto program = decoder.decode(is);

    std::vector<std::string> initialSymbols({"0"});
    Tape tape(initialSymbols);

    Simulator sim(std::move(program), tape);

    while (sim.step())
    {
    }

    std::vector<std::string> expected({""});

    return sim.getState() == 2 && tape.getAllSymbols() == expected;
}
```

Likewise, the next test will be for the input `"1"`. Here we expect the tape to end up with a single cell containing the symbol `"A"`.

src/tests/simulator-tests.cc

```cpp
static bool stringOf1ATest(void)
{
    std::stringstream is(std::string(
    "\x54\x55\x52\x49\x4E\x47\x09\x00\x31\x00\x30\x00\x01\x01\x00\x30"
    "\x00\x31\x00\x00\x00\x00\x00\x00\x01\x02\x02\x31\x00\x00\x01\x02"
    "\x01\x30\x00\x30\x00\x01\x01\x01\x31\x00\x31\x00\x01\x01\x01\x41"
    "\x00\x41\x00\x01\x01\x01\x00\x41\x00\x00\x00\x00\x41\x00\x41\x00"
        "\x00\x00",
        66));
    Decoder decoder;
    auto program = decoder.decode(is);

    std::vector<std::string> initialSymbols({"1"});
    Tape tape(initialSymbols);

    Simulator sim(std::move(program), tape);

    while (sim.step())
    {
    }

    std::vector<std::string> expected({"A"});

    return sim.getState() == 2 && tape.getAllSymbols() == expected;
}
```

And finally, we will test the input 5, which this program expects in binary, with one cell per digit (`"1"`, `"0"`, `"1"`). The expected output is 5 cells, each containing the symbol `"A"`.

src/tests/simulator-tests.cc

```
static bool stringOf5AsTest(void)
{
    std::stringstream is(std::string(
        "\x54\x55\x52\x49\x4E\x47\x09\x00\x31\x00\x30\x00\x01\x01\x00\x30"
        "\x00\x31\x00\x00\x00\x00\x00\x00\x01\x02\x02\x31\x00\x00\x01\x02"
        "\x01\x30\x00\x30\x00\x01\x01\x01\x31\x00\x31\x00\x01\x01\x01\x41"
        "\x00\x41\x00\x01\x01\x01\x00\x41\x00\x00\x00\x00\x41\x00\x41\x00"
        "\x00\x00",
        66));
    Decoder decoder;
    auto program = decoder.decode(is);

    std::vector<std::string> initialSymbols({"1", "0", "1"});
    Tape tape(initialSymbols);

    Simulator sim(std::move(program), tape);

    while (sim.step())
    {
    }

    std::vector<std::string> expected({"A", "A", "A", "A", "A"});

    return sim.getState() == 2 && tape.getAllSymbols() == expected;
}
```

To tie these tests together, we will implement this component's `allTests` method.

src/tests/simulator-tests.cc

```
void tests::simulator::allTests(tests::Record &record)
{
    RUN_TEST(emptyStringOfAsTest, record);
    RUN_TEST(stringOf0AsTest, record);
    RUN_TEST(stringOf1ATest, record);
    RUN_TEST(stringOf5AsTest, record);

    return;
}
```

To make sure that these tests are run, we must add this component to the `FOR_ALL_COMPONENTS` macro in `src/tests/tests.hh`.

```
src/tests/tests.hh
```

```
#define FOR_ALL_COMPONENTS \
    X(encoder)          \
    X(decoder)          \
    X(vlq)              \
    X(simulator)
```

Finally, we must add our new source files and dependency targets to the `tests` target in the Makefile.

```
Makefile
```

```
tests: vlq encoder decoder tape simulator
       $(CC) $(FLAGS) -o dist/tests dist/encoder.o \
                                     dist/decoder.o \
                                     dist/vlq.o \
                                     dist/tape.o \
                                     dist/simulator.o \
                                     src/tests/vlq-tests.cc \
                                     src/tests/encoder-tests.cc \
                                     src/tests/decoder-tests.cc \
                                     src/tests/simulator-tests.cc \
                                     src/tests/tests.cc
```

Now, when we run our test suite, all of our tests should pass, indicating that our simulator is working as expected.

```
Running encoder tests:
  simpleTest: ✅
PASSED: 1/1 ✅

Running decoder tests:
  simpleTest: ✅
PASSED: 1/1 ✅

Running vlq tests:
  is0EncodedCorrectly: ✅
  is50EncodedCorrectly: ✅
  is123456789EncodedCorrectly: ✅
  is0DecodedCorrectly: ✅
  is50DecodedCorrectly: ✅
  is123456789DecodedCorrectly: ✅
  is123456789DecodedCorrectlyWithMoreBytes: ✅
PASSED: 7/7 ✅

Running simulator tests:
  emptyStringOfAsTest: ✅
  stringOf0AsTest: ✅
  stringOf1ATest: ✅
  stringOf5AsTest: ✅
PASSED: 4/4 ✅

OVERALL PASSED: 13/13 ✅
```

4.4 The TMVM CLI

In this section we will be creating a command-line interface to the simulator. The usage will be as follows:

```
tmvm <program> [<symbol0> <symbol1> <symbol2> ...]
```

In other words, the first argument will be a path to a file containing the machine code of the program to run. The next arguments are optional and there can be arbitrarily many of them. They each represent a symbol to go on the tape, from left to right. As usual, the convention will be that the machine starts with the read/write head over the rightmost cell from the input.

We will write this code in `src/tmvm.cc`. The first part of the `main` method will check whether too few arguments have been specified, and if so, print the usage message, and exit.

src/tmvm.cc

```cpp
#include <iostream>
#include <fstream>
#include "decoder.hh"
#include "simulator.hh"

int main(int argc, char **argv)
{

    if (argc < 2)
    {
        std::cout << "Usage: " << argv[0] << " <program> [<symbol0>
 ↪  <symbol1> <symbol2> ...]" << std::endl;
        return 1;
    }
```

Then, we will open the file specified by the first argument. If that file could not be found or was in some other way problematic, we will print out another error message to that effect, and exit.

src/tmvm.cc

```
    std::ifstream is(argv[1]);

    if (!is.is_open())
    {
        std::cout << "Could not find program \"" << argv[1] << "\""
 ↪  << std::endl;
        return 2;
    }
```

Next, we will instantiate a decoder, and attempt to decode the program. If this fails (meaning that the `Decoder::decode` method returned a **`nullptr`**), then print out yet another error message, and exit. Since we now have a pointer to the decoded program, we can close the input file.

src/tmvm.cc

```
    Decoder decoder;
    auto program = decoder.decode(is);
    if (!program)
    {
        std::cout << "Failed to decode program: \"" << argv[1] <<
 ↪  "\"" << std::endl;
        return 3;
    }

    is.close();
```

Next, we initialise a tape using the remaining arguments to the program, if any exist.

src/tmvm.cc

```
    std::vector<std::string> initialSymbols;
    initialSymbols.assign(argv + 2, argv + argc);
    Tape tape(initialSymbols);
```

We can now construct a simulator using the program pointer and the tape. We will call its `step` method repeatedly until it returns `false` (which will indicate that the program has finished running).

src/tmvm.cc

```
    Simulator sim(std::move(program), tape);

    while (sim.step())
    {
    }
```

Finally, we will output the simulator, and exit.

src/tmvm.cc

```
    std::cout << sim << std::endl;

    return 0;
}
```

We can add this as a new target to the Makefile.

Makefile

```
tmvm: dist-dir
	$(CC) $(CFLAGS) -o dist/tmvm dist/decoder.o \
	                           dist/vlq.o \
	                           dist/tape.o \
	                           dist/simulator.o \
	                           src/tmvm.cc
```

Furthermore, we will add a target called `all`, which will compile all of the components (excluding the tests).

Makefile

```
all: vlq encoder decoder tape simulator tmvm
```

To run the TMVM, we will need to have a file which contains the machine code for the program we want to execute. Again using the example from page 29, that machine code would be the following bytes:

```
54 55 52 49 4E 47 09 00 31 00 30 00 01 01 00 30 00 31 00 00 00 00
 ↪  00 00 01 02 02 31 00 00 01 02 01 30 00 30 00 01 01 01 31 00 31
 ↪  00 01 01 01 41 00 41 00 01 01 01 00 41 00 00 00 00 41 00 41 00
 ↪  00 00
```

We could store these bytes in a file called `programs/string_of_As.tur`, with the `.tur` extension indicating that this file is a Turing machine program.

To build the TMVM and run this program on it with input 20 (10100 in binary), we would run the following commands:

```
make
./dist/tmvm ./programs/string_of_As.tur 1 0 1 0 0
```

And we would see the output:

```
State: 2

| A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A |
  ↑

| A | A | A | A |
```

4.5 Recovering the State Strings

At the moment the TMVM is a fully functioning Turing Machine simulator. That is to say, any program which can be run on a Turing machine can be encoded into our machine code format and then executed on the TMVM. However, there are still some features left to implement.

One of these features (which will be particularly helpful when writing the pseudo-assembler) is the ability to convert our state names from integers back into the strings we used when giving our program to the encoder.

Of course, the simulator doesn't have access to this information, as it was discarded after the encoding process. Nonetheless, we can add it back in in the form of metadata.

As a recap, the machine code of a Turing machine program consists of the magic bytes, the number of rules, and then a chunk of machine code for each rule. We can add some additional data after this to encode the mapping from state indices to state strings.

This will be optional, and the program will run as intended even if no metadata is present.

Since there will eventually be multiple types of metadata, we can't simply output the state strings directly after the machine code for the rules. Instead, we will first output a single byte representing what type of metadata is about to follow. Of course, we could instead use a VLQ, but 256 different types of metadata will be more than enough for our purposes.

We will create a new header file called `src/metadata.hh` which will contain an enum mapping metadata types to chars.

src/metadata.hh

```
#pragma once

enum MetadataType : char
{
    STATE_STRINGS = 0
};
```

For now this enum only has one value, but we denote the underlying type as `char` to remind us to never exceed the limit of 256 types of metadata.

In the encoder we can add a new method to output the state string metadata. We will declare this method in the `src/encoder.hh` header.

src/encoder.hh

```
    void addStateStrings(std::ostream &);
```

This method will be optionally called after the `Encoder::output` method, and will output the byte representing the metadata type (in this case, a null byte) followed by all of the state strings in order of state index (i.e., the string for state **`0`** followed by the string for state **`1`**, etc.).

To achieve this, the encoder must be able to get the state string for a given index. Unfortunately, the `m_stateNames` hash table only maps in the other direction. To solve this, we will add in another data structure, `m_indexToState`, which will be a vector containing all of the state names in the correct order.

src/encoder.hh

```
    std::vector<std::string> m_indexToState;
```

In `encoder.cc`, we can modify the constructor and the `getStateIndex` methods so that they keep this vector up-to-date.

src/encoder.cc

```
Encoder::Encoder(void)
{
    m_stateNames["start"] = 0;
    m_indexToState.push_back("start");
    m_stateCount = 1;
}

state_t Encoder::getStateIndex(const std::string &name)
{
    auto search = m_stateNames.find(name);
    if (search == m_stateNames.end())
    {
        state_t index = m_stateCount++;
        m_stateNames[name] = index;
        m_indexToState.push_back(name);
        return index;
    }
    return search->second;
}
```

As you can see, whenever a new state index is allocated, the associated name is added to `m_indexToState`.

Now we can implement the `addStateStrings` method, which is fairly simple.

src/encoder.cc

```
#include "metadata.hh"
```

src/encoder.cc

```
void Encoder::addStateStrings(std::ostream &os)
{
    os << (char) MetadataType::STATE_STRINGS;
    for (auto state : m_indexToState)
    {
        os << state << '\0';
    }
}
```

Note that we output a null byte after every state string. This is so that when we decode this program, the decoder will be able to determine where one state string ends and the next begins. It is unrelated to the fact that a null byte is also used to indicate that the following metadata encodes state strings.

We can confirm that this is working as expected by adding a unit test for the encoder. We will call it `simpleTestWithStateStrings` and it will be very similar to `simpleTest`, except that we invoke the `Encoder::addStateStrings` method, and the expected output includes a null byte followed by the state strings at the end.

src/tests/encoder-tests.cc

```
static bool simpleTestWithStateStrings(void)
{
    Encoder encoder;
    encoder.addRule("start", "1", "0", Direction::RIGHT,
 ↪  "go_to_end");
    encoder.addRule("start", "0", "1", Direction::LEFT, "start");
    encoder.addRule("start", "", "", Direction::RIGHT, "done");
    encoder.addRule("done", "1", "", Direction::RIGHT, "done");
    encoder.addRule("go_to_end", "0", "0", Direction::RIGHT,
 ↪  "go_to_end");
    encoder.addRule("go_to_end", "1", "1", Direction::RIGHT,
 ↪  "go_to_end");
    encoder.addRule("go_to_end", "A", "A", Direction::RIGHT,
 ↪  "go_to_end");
    encoder.addRule("go_to_end", "", "A", Direction::LEFT, "start");
    encoder.addRule("start", "A", "A", Direction::LEFT, "start");

    std::ostringstream os;

    encoder.output(os);
    encoder.addStateStrings(os);

    auto result = stringToHex(os.str());

    std::string expected =
        "545552494E4709003100300001010030"
        "0031000000000000010202310000010 2"
        "0130003000010101310031000101014 1"
        "0041000101010041000000000410041 00"
        "00000073746172740 0676F5F746F5F65"
        "6E6400646F6E6500";

    return result == expected;
}

void tests::encoder::allTests(tests::Record &record)
{
    RUN_TEST(simpleTest, record);
    RUN_TEST(simpleTestWithStateStrings, record);

    return;
}
```

Now that the encoder can handle state strings, we will likewise update the decoder.

We will give the decoder a private method called `parseStateStrings`, which will accept as parameters references to both the input stream and the `Program` object currently being constructed.

```
src/decoder.hh

private:
    void parseStateStrings(std::istream &, Program &);
```

This method is private because we do not need to invoke it manually when invoking the decoder, but it will instead be called automatically when the `decode` method detects that the state name metadata is present. In fact, we will create a separate such private method for each type of metadata we will implement.

In this spirit, we will modify the `decode` method to do just that. Once we've finished decoding each of the rules, the file index will either be just before the end of the file (i.e., there is no metadata) or will be just before some other byte (i.e., there is metadata). In the latter case, the aim is to parse this metadata and then repeat until there is no metadata remaining. We are assuming here that there may be multiple types of metadata present in any order.

We can achieve this with the following loop before the return statement:

```
src/decoder.cc

#include "metadata.hh"
```

```
src/decoder.cc

    char c = is.get();
    while (!is.eof())
    {
        switch (static_cast<MetadataType>(c))
        {
        case STATE_STRINGS:
            parseStateStrings(is, program);
            break;
        default:
            // Unknown metadata type
            break;
        }
        c = is.get();
    }
```

This is building the framework which will allow the decoder to handle many different types of metadata by simply adding more cases to the switch statement.

In the case that metadata is present, the `parseStateStrings` method is called. We need to decide where it will store all of the state strings, and the natural choice is to store it in the `Program` struct.

We will add two fields to this struct. The first is a boolean flag indicating whether or not state strings have been decoded for this program. The second will be a vector containing them if they do exist. If they don't, this vector will remain empty.

src/program.hh

```
    bool hasStateStrings{false};
    std::vector<std::string> stateStrings;
```

Now, we can go back to `decoder.cc` and implement the `parseStateStrings` method.

src/decoder.cc

```
void Decoder::parseStateStrings(std::istream &is, Program &program)
{
    program.hasStateStrings = true;
    program.stateStrings.resize(program.numStates);

    std::string stateStringBuffer;
    for (state_t i = 0; i < program.numStates; i++)
    {
        std::getline(is, stateStringBuffer, '\0');
        program.stateStrings[i] = stateStringBuffer;
    }
}
```

This method simply sets the `hasStateStrings` flag for the program, resizes the `stateStrings` vector to the number of states (which is already known from earlier stages of the decoding process), and then reads that number of null-terminated strings from the input stream, storing them in the `stateStrings` vector. Subsequently, the value stored at `program.stateStrings[i]` will be the state string for state `i`.

We will add a unit test for the decoder, but first we must modify the existing one, `simpleTest` in `src/tests/decoder-tests.cc`. This test should now also ensure that the `hasStateStrings` flag is unset for the program it decodes, because the input machine code does not contain any metadata.

```
src/tests/decoder-tests.cc
```

```cpp
    if (program->hasStateStrings)
        return false;
```

Now we can add a new test, `simpleTestWithStateStrings`, which is similar to `simpleTest` except that the input machine code will include the state name metadata, we will ensure that the `program.hasStateStrings` flag is set rather than unset, and we will check whether the contents of the `program.stateStrings` vector are the state strings we expect.

src/test/decoder-tests.cc

```
static bool simpleTestWithStateStrings()
{
    std::stringstream is(std::string(
        "\x54\x55\x52\x49\x4E\x47\x09\x00\x31\x00\x30\x00\x01\x01\x00\x30"
        "\x00\x31\x00\x00\x00\x00\x00\x00\x01\x02\x02\x31\x00\x00\x01\x02"
        "\x01\x30\x00\x30\x00\x01\x01\x01\x31\x00\x31\x00\x01\x01\x01\x41"
        "\x00\x41\x00\x01\x01\x01\x00\x41\x00\x00\x00\x00\x41\x00\x41\x00"
        "\x00\x00\x00\x73\x74\x61\x72\x74\x00\x67\x6F\x5F\x74\x6F\x5F\x65"
        "\x6E\x64\x00\x64\x6F\x6E\x65\x00",
        88));

    Decoder decoder;
    auto program = decoder.decode(is);

    if (!program)
        return false;
    if (program->rules.size() != 9)
        return false;
    if (program->numStates != 3)
        return false;
    if (!(program->hasStateStrings))
        return false;
    if (program->stateStrings != std::vector<std::string>({"start", "go_to_end", "done"}))
        return false;
    if (!(program->rules[0] == Rule{0, "1", "0", Direction::RIGHT, 1}))
        return false;
    if (!(program->rules[1] == Rule{0, "0", "1", Direction::LEFT, 0}))
        return false;
    if (!(program->rules[2] == Rule{0, "", "", Direction::RIGHT, 2}))
        return false;
    if (!(program->rules[3] == Rule{2, "1", "", Direction::RIGHT, 2}))
        return false;
    if (!(program->rules[4] == Rule{1, "0", "0", Direction::RIGHT, 1}))
        return false;
    if (!(program->rules[5] == Rule{1, "1", "1", Direction::RIGHT, 1}))
        return false;
    if (!(program->rules[6] == Rule{1, "A", "A", Direction::RIGHT, 1}))
        return false;
    if (!(program->rules[7] == Rule{1, "", "A", Direction::LEFT, 0}))
        return false;
```

The encoder and decoder now support state string metadata, and the state strings are stored in the `Program` object. We can use this in the simulator to make our lives easier when it comes to debugging. For example, in the `Simulator` output method, normally the current state index is printed. We can modify this method to use the state string if it exists.

src/simulator.cc

```cpp
std::ostream &operator<<(std::ostream &os, const Simulator &sim)
{
    os << "State: ";
    if (sim.m_program->hasStateStrings)
        os << sim.m_program->stateStrings[sim.m_state];
    else
        os << sim.m_state;
    os << std::endl;
    return os << sim.m_tape;
}
```

Now, when we run our tests, we should see that they all pass:

```
Running encoder tests:
  simpleTest: ✅
  simpleTestWithStateStrings: ✅
PASSED: 2/2 ✅

Running decoder tests:
  simpleTest: ✅
  simpleTestWithStateStrings: ✅
PASSED: 2/2 ✅

Running vlq tests:
  is0EncodedCorrectly: ✅
  is50EncodedCorrectly: ✅
  is123456789EncodedCorrectly: ✅
  is0DecodedCorrectly: ✅
  is50DecodedCorrectly: ✅
  is123456789DecodedCorrectly: ✅
  is123456789DecodedCorrectlyWithMoreBytes: ✅
PASSED: 7/7 ✅

Running simulator tests:
  emptyStringOfAsTest: ✅
  stringOf0AsTest: ✅
  stringOf1ATest: ✅
  stringOf5AsTest: ✅
PASSED: 4/4 ✅
```

```
OVERALL PASSED: 15/15 ✅
```

And when we save our machine code with state string metadata included into a file, say `programs/string_of_As_with_state_strings.tur`, and run it with the TMVM,

```
make
./dist/tmvm ./programs/string_of_As_with_state_strings.tur 1 0 1
```

we see that the state is printed as its string, **`done`**, rather than its index, **`2`**.

```
State: done
-----------------------
| A | A | A | A | A |
-----------------------
  ↑
```

5 The Turing Machine Pseudo-Assembly Language — TASM

5.1 What is Pseudo-Assembly?

In previous chapters we outlined the structure of a machine code for our Turing machine programs. Though efficiently stored and fast for a simulator to parse, it would be very difficult and time-consuming to write by hand. There is a sense in which machine code, by definition, is stored as raw bytes — devoid of any mnemonics, comments, or human-friendly metaphors.

Hand-written machine code would be prone to bugs, and would resist interpretation by collaborators. This is a problem which affects almost all machine code languages, and so each often has its own accompanying “assembly” language. These languages consist of simple instructions (typically a one-to-one mapping with machine code instructions) but are written in human-friendly ASCII, rather than raw bytes.

For example, in the RISC-V architecture, the instruction which adds together the values stored in registers `t0` and `t1`, storing the result in register `t2` is written in machine code as follows:

```
0x006283B3
```

Whereas the equivalent assembly instruction is:

```
ADD t2, t0, t1
```

Clearly the latter is much easier for humans to work with. A program called an “assembler” is used to convert assembly code to machine code.

In this sense, our own machine code specification already has an assembly language — the Δ-function notation. This, like an assembly language, lets us specify our rules in a more human-friendly format while having a one-to-one correspondence with the machine code rules. Likewise, the encoder component we’ve created acts like an assembler.

However, one key way in which the Δ-function notation is different from a traditional assembly language is that the rules are not executed sequentially. In fact, the rules can be specified in any order at all and the execution of the program will be identical. This makes it difficult to use this language to implement algorithms where sequentiality is the underlying paradigm.

If our goal is eventually to create a high-level C-like programming language for the Turing machine, it will be exceedingly helpful to design a simpler assembly-like intermediate language, or "pseudo-assembly".

Though this will not be an assembly language in a technical sense, as it will lack a one-to-one correspondence between its own instructions and the Turing machine rules, it will look a lot like an assembly language, being made up of a list of simple instructions which execute sequentially.

This is a very important shift in the way programs will be written for the Turing machine. Writing machine code directly (or indeed, using Δ-function notation) forces the programmer to think in terms of **rules**, which either apply or don't apply in any given situation. Whereas, writing programs in this pseudo-assembly language will allow the programmer to think in terms of **instructions**, which are executed in sequence.

The goal of this chapter is to write a tool which can bridge this gap, converting pseudo-assembly into machine code.

Let's clear up some terminology. Though, as mentioned above, this language ("TASM", or *Turing Assembly*) will not meet the technical definition of an assembly language for this machine, we will nonetheless refer to it as an assembly language from here on. I feel that this is appropriate for several reasons:

- TASM will look like other assembly languages.
- Programming in TASM will require a very similar mindset to programming in other assembly languages.
- This convention will allow us to refer to the conversion program as an "assembler" rather than the more generic term, "compiler". This will avoid confusion between it and the compiler for the C-like language we will create later.

To get a better feel for the assembly language we're trying to create, let's write an example program.

Suppose we wanted to add together the numbers 15 and 276. The TASM instructions might look like:

```
PushInt 15
PushInt 276
Add
```

We also want to allow jumping between instructions. We can achieve this by treating lines which start with a ":" character to be "labels", and use an instruction such as `Goto` to continue executing the program from that label.

For example, the program below will count upwards forever in an infinite loop.

```
PushInt 0
:loop
PushInt 1
Add
Goto loop
```

We also want to allow code comments. For example, we can ignore anything which follows a ";" character, up until the next newline.

```
PushInt 0
:loop
PushInt 1
Add
Goto loop       ; Jumps back to the start of the loop
```

5.2 What Goes on the Tape?

As you may have guessed from the example TASM programs in the previous section, we are going to use the Turing machine as a stack machine.

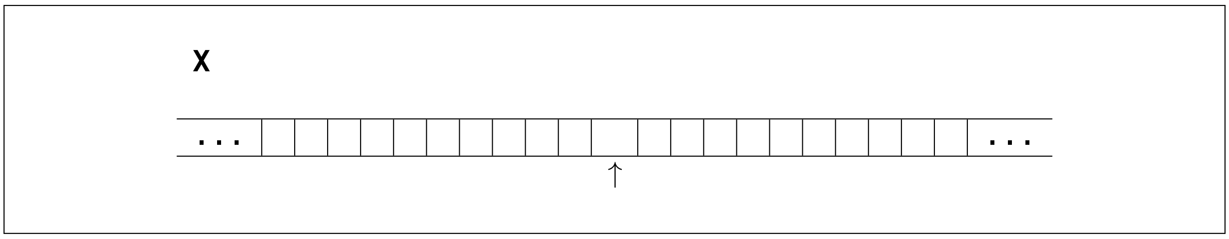
A stack is a data structure with two main operations:

- `Push`, which adds an item to the end of the stack.
- `Pop`, which removes an item from the end of the stack.

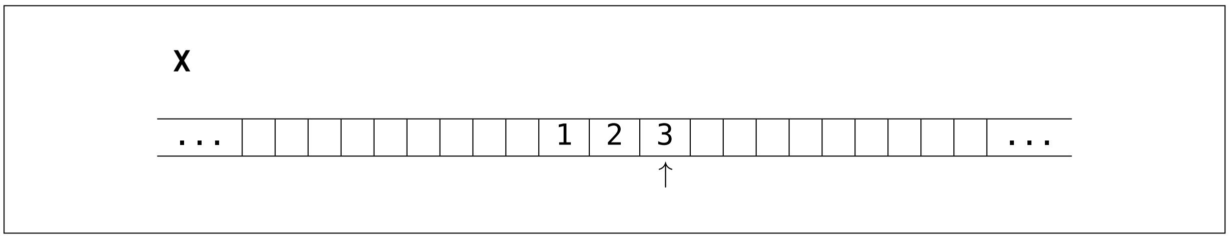
A stack is a *LIFO* data structure (*last in, first out*). This means that when you pop an item off of the stack, you remove the most recently pushed item.

We could imagine using the tape of the Turing machine as a stack where each pushed element is placed on the right-han d side of the most recently pushed element, and then elements are popped by removing the right-most element from the tape.

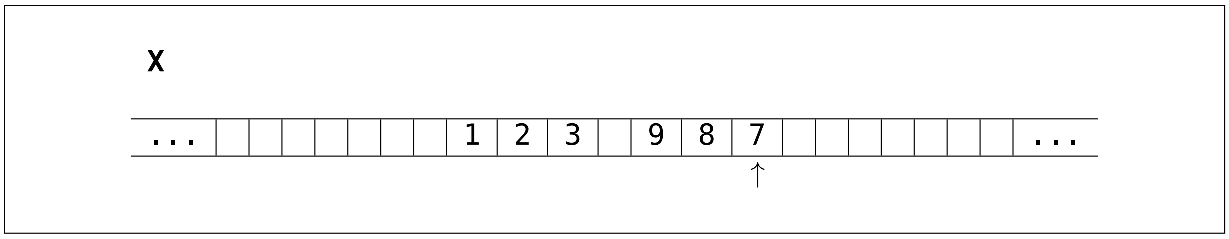
So, for example, the stack being empty might look like an empty tape.



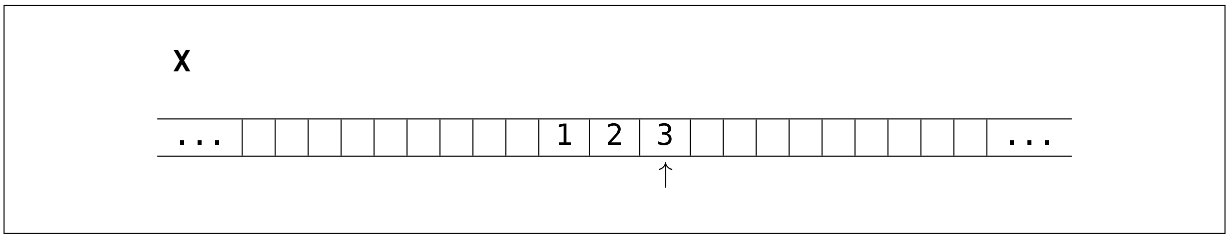
If the number 123 is pushed onto the stack, the tape might look like this:



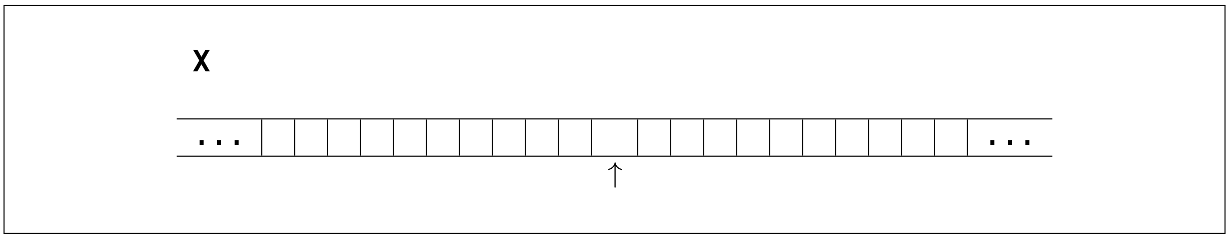
If the number 987 is pushed afterwards, the tape might look like this:



Then, when a pop occurs, the number 987 (the most recently pushed value) would be removed.



Furthermore, when another pop occurs, the 123 would be removed, leaving the stack (and the tape) once again empty.



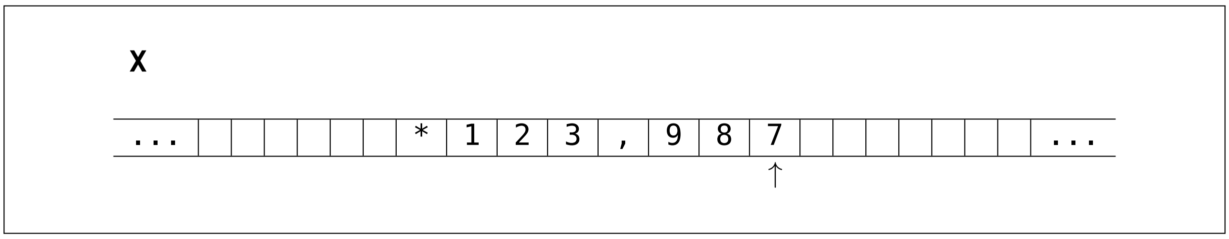
We will not use our tape exactly like this, as the assembler will be easier to implement if we make a few changes.

The first change will be to use a marker symbol (`*`) in the cell immediately to the left of the stack to indicate where the stack begins. We will call this symbol the “stack marker”.

We will also have a separator symbol (`,`) in between each element, rather than an empty cell. We will call this symbol the “stack separator”.

The choice to use these particular symbols was arbitrary.

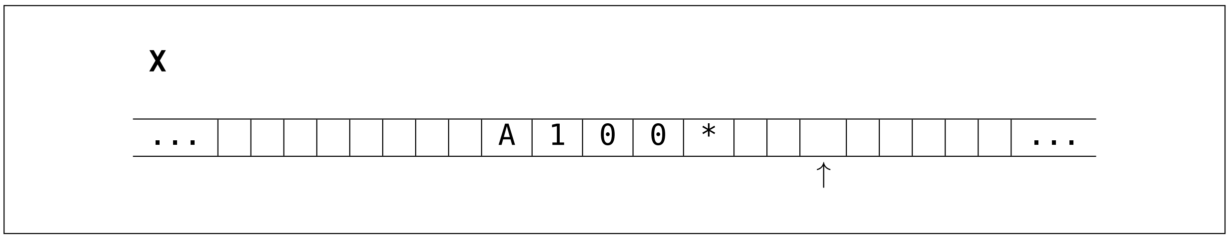
With these changes, the stack containing 123 and 987 will look as follows:



On the left of the stack marker will be an area called the “environment”. This is where we will store named variables.

If there are no variables stored, then all of the cells to the left of the stack marker will be empty, as in the diagram above.

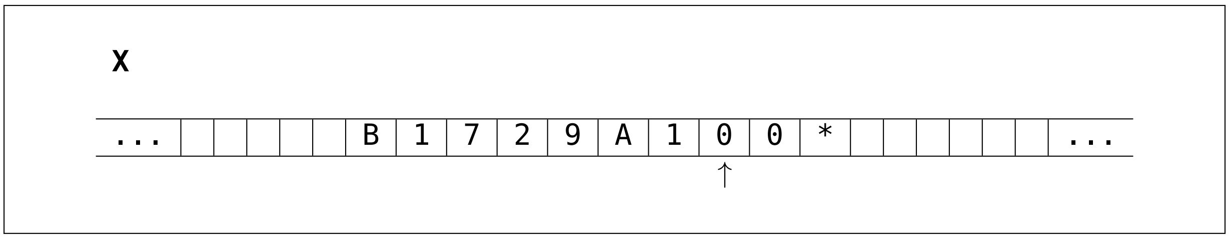
If there were, for example, a variable named `A` with a value of 100, then it would be stored as follows:



Note that in the diagram above that the stack is shown as empty, although there may well be values in the stack as well as the environment.

Note also that the variable name `A` is its own symbol. This applies even if the variable name is multiple characters long. This might imply that a program which uses arbitrarily many variables might result in the tape containing arbitrarily many symbols, and this is true. However, this is not a problem because the complete set of symbols will certainly be finite for any given program and can be known at compile-time.

If in addition to `A`, another variable `B` was stored on the environment with a value 1729, the tape would look as follows:



Note that unlike with the stack, we do not need a separator symbol between variables stored in the environment. This is because, as long as we do not allow variable names to be numeric digits (or any symbol which can occur in a value), then the variable names themselves acts as separators of sorts.

The Turing machine rules which execute a given TASM instruction will vary wildly depending on where exactly the read/write head starts off, and what the tape looks like. It is therefore crucial that we maintain some invariants:

1. Once any TASM instruction has finished executing, the read/write head must be over the rightmost non-empty cell of the stack. If the stack is empty, then the read/write head must be over the stack marker.

2. Once any TASM instruction has finished executing, the stack and environment must be well-formed. For example, there must be no gaps or unexpected symbols in either.

Though these conditions may (and indeed will) be broken in the course of executing some of the TASM instructions, they must be repaired by the time the instruction finishes.

There are one or two exceptions to these invariants but we will discuss this more when we implement the offending TASM instructions.

For the most part, these invariants will allow us to make some key assumptions about the machine's situation when each new TASM instruction executes.

Something to note here is that the numbers stored in the tape are written in base 10, unlike traditional computers which work in binary. While the assembler could indeed be programmed to make the machine work in any base, we will choose base 10 for two reasons.

The first reason is that base 10 is the number system that most of us will be most comfortable with reading. Debugging this project will be enough of a nightmare as it is, and there is no need to make it harder on ourselves by having to convert between some other base and decimal in our heads while trying to figure out which numbers are stored in the tape.

The other, and perhaps more practical reason comes from the fact that the tape is not random access in the way that memory on most computers is. This is to say that if we want to fetch a value which is far away from the read/write head, it takes a lot of time to travel all the way over to it. As a result of this, having numbers take up fewer cells means that the read/write head will tend to have a shorter journey to most destinations.

This would suggest that a high base is desirable to make the machine run faster, so why stop at 10? Well, the more possible symbols which can exist in a number, the more rules will have to be defined for states to handle seeing those symbols, leading to an overall bigger program.

Base 10 strikes an appropriate balance between speed, machine code minimisation, and readability.

As well as numbers, several other data types will be allowed as values on the stack, and as variable values in the environment. A full list of allowed value types is:

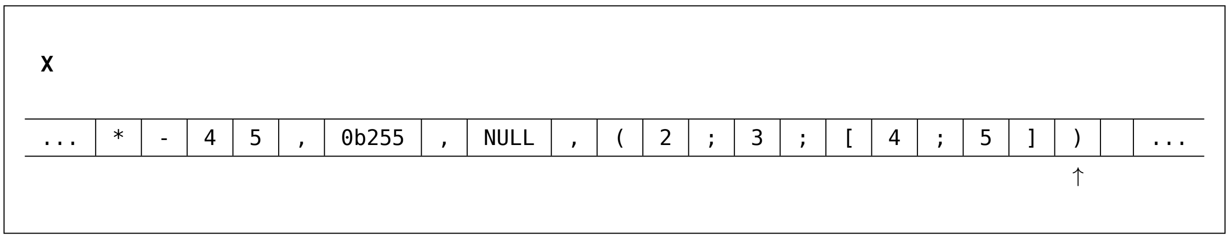
- Integers. These are made of consecutive cells each containing a symbol from `0` to `9`. They may optionally have the symbol `-` in the leftmost cell, indicating that the integer is negative. Leading `0`s are not permitted. The number 0 will be represented by a single cell with the symbol `0` (i.e., no extra leading zeros, and no minus sign).

- Octets. There are 256 possible octet values. Each one consists of a single cell whose symbol is `0d` followed by the octet's value in decimal. For example, the octet `0x41` (`'A'` in ASCII) would have the symbol `0d65`.

- Null. This is a single cell with the symbol `NULL`.

- Arrays. These span multiple cells and contain other valid values. The leftmost cell contains the symbol `(`. This is them followed by the cells of arbitrarily many other values, separated by a cell with a symbol `;`. These values are the "elements" of the array. After this, there is the symbol `)`. Note that if any of the elements of the array is itself an array, the inner arrays' `(` and `)` cells are replaced with `[` and `]` cells respectively.

For example, if the stack contained the values -45, the octet `0xFF`, a null value, and an array with three elements — 2, 3, and another array containing the numbers 4 and 5 — the tape would look like this:


X
... | * | - | 4 | 5 | , | 0b255 | , | NULL | , | (| 2 | ; | 3 | ; | [| 4 | ; | 5 |] |) | | ...
↑

These specifications might seem a bit complicated and arbitrary, but when we come to implement the TASM instructions hopefully the merits of this design will become clear.

5.3 Pure Instructions

While computationally very powerful, a Turing machine leaves a lot to be desired when it comes to user interfaces. Limiting ourselves to simply an input and an output with no way to interact with the program while it's running would be a bit disappointing.

Later in the chapter we will discuss ways to hack in various I/O systems to the TMVM, but some may rightfully object that such modifications are cheating, as they would not exist on a Turing machine as we defined it earlier.

To rectify this, we will split our TASM instructions into two sets: "Pure" and "impure". The pure instructions are those which would work on a Turing machine with no modifications.

A list of the pure instructions we will implement (further split into subcategories) is below.

5.3.1 Memory operations

Instruction	Description
`PushInt [i]`	Pushes integer `[i]` to the stack
`PushOctet [o]`	Pushes octet `[o]` to the stack
`PushNull`	Pushes a null value to the stack
`PushInstNo`	Pushes the index of the currently executing TASM instruction to the stack
`Duplicate`	Duplicates the last value on the stack
`Pop`	Removes the last value from the stack
`Load [VAR]`	Looks up variable `[VAR]` in the environment and pushes its value to the stack
`Set [VAR]`	Adds a variable `[VAR]` to the environment whose value is the last element on the stack. If `[VAR]` is already defined in the environment, adds another definition
`Unset [VAR]`	Removes the most recent definition of `[VAR]` from the environment

5.3.2 Array operations

Instruction	Description
`Split`	If the last element on the stack is an array, split its elements into separate values on the stack
`Join`	Joins n values from the stack into an array, starting at the second-to-last value, where n is the last value.

5.3.3 Mathematical operations

Instruction	Description
`Add`	Adds the last two integers on the stack, pushing the value onto the stack
`Negate`	Multiplies the last integer on the stack by -1
`Lshift`	Adds n trailing zeros to the second-last integer p on the stack, where n is the last value on the stack. If $p = 0$, then p will remain 0
`Rshift`	Removes the n least-significant digits from the second-last integer p on the stack, where n is the last value on the stack. If p has fewer than $n+1$ digits (not including the sign symbol), then p will become 0

5.3.4 Type operations

Instruction	Description
`IsNull`	If the last value on the stack is null, push 1, else push 0
`IsInt`	If the last value on the stack is an integer, push 1, else push 0
`IsOctet`	If the last value on the stack is an octet, push 1, else push 0
`IsArray`	If the last value on the stack is an array, push 1, else push 0
`IntToOctetArray`	If the last value on the stack is an integer, replace it with an array containing all of the corresponding ASCII octets of the integer followed by the number of such octets
`OctetArrayToInt`	If the last value on the stack is an array containing n octets between `0d48` and `0d57` inclusive (the first octet is also allowed to be `0d45`) followed by the integer n, then replace the array with the integer spelled out by the octets in ASCII

5.3.5 Branching operations

Instruction	Description
`Goto [label]`	Continue executing the program from the instruction immediately following label `[label]`
`Jump [i]`	Jump forwards by `[i]` instructions. Note: `[i]` may be negative
`JumpZero [i]`	Jump forwards by `[i]` instructions if and only if the last value on the stack is 0. Note: `[i]` may be negative
`JumpNotZero [i]`	Jump forwards by `[i]` instructions if and only if the last value on the stack is not 0. Note: `[i]` may be negative
`Return`	Continue executing from the instruction whose index is equal to the last value on the stack