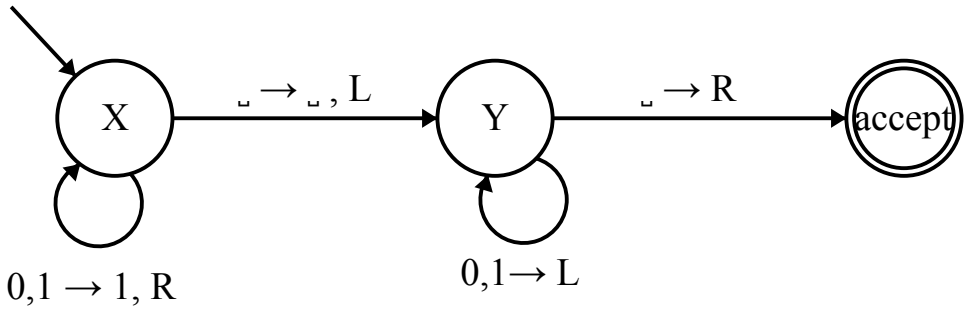
Turing Machines

I wrote a little program to implement Turing Machines. The first step was to make a `Tape` class that supports the operations `read`, `write`, and `move`. The `move` method has a parameter `direction` that can be `LEFT`, `RIGHT`, or `STAY`. The `Tape` is implemented with a doubly-linked list in order to simulate infinite length in both directions.

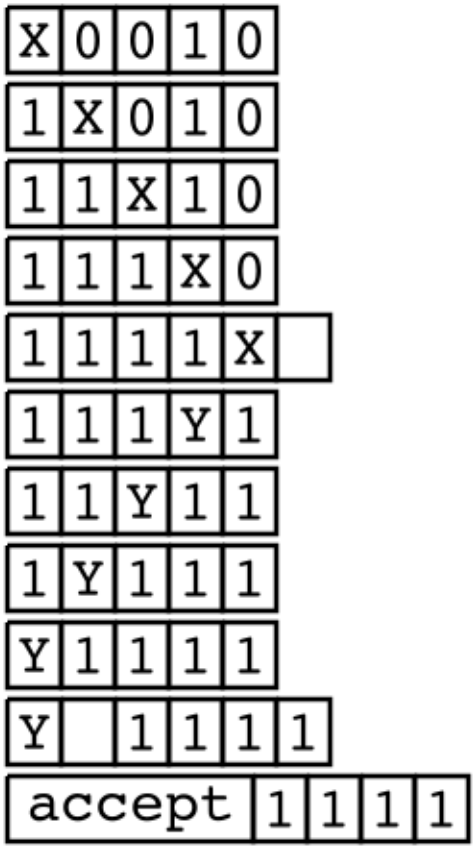
Armed with this `Tape` data structure, implementing the Turing Machine is quite easy. Here it is.

```
class TuringMachine:
    def __init__(self, start, transition):
        self.transition = transition
        self.start = start

    def compute(self, input):
        self.tape = Tape(input)
        self.state = self.start

    def step(self):
        (state, symbol, motion) = self.transition(self.state, self.tape.read())
        self.state = state
        self.tape.write(symbol)
        self.tape.move(motion)

    def config(self):
        return self.tape.leftofhead() + [self.state] + self.tape.rightofhead()
```

To initialize a new `TuringMachine` object, you give it a start state and the transition function. It assumes that any symbols that appear on the tape are tape symbols as are any symbols that the transition function says to write. It also assumes that there are states called `'accept'` and `'reject'`. Any state that comes out of the transition function is considered a real state.

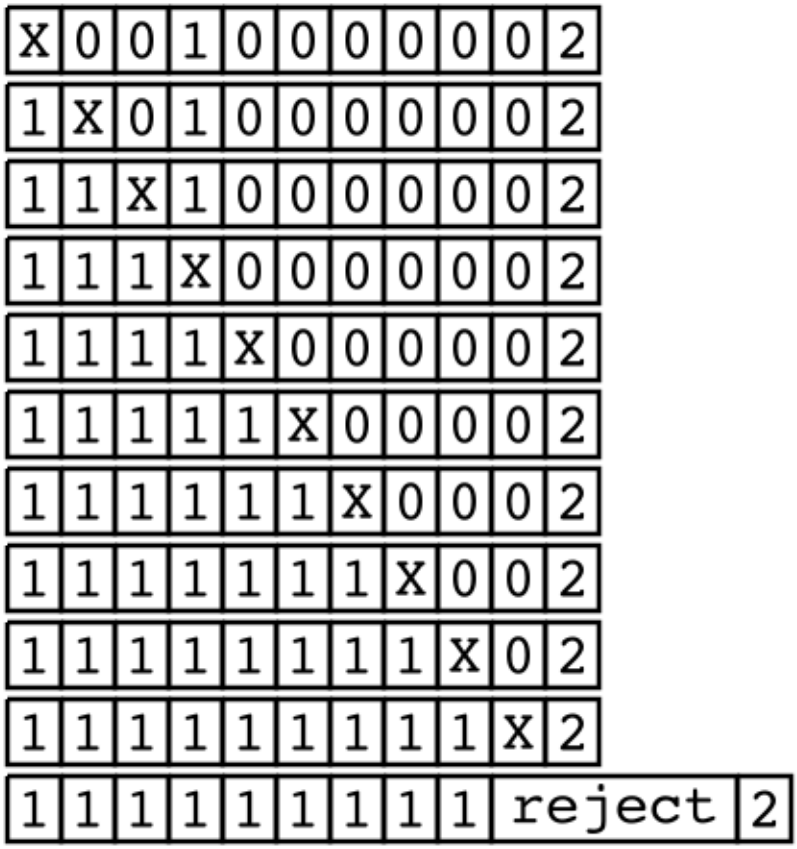



```
def trans(q, a):
    if q == 'X':
        if a in ['0', '1']:
            return ('X', '1', RIGHT)
        if a == ' ':
            return ('Y', ' ', LEFT)
    if q == 'Y':
        if a in ['0', '1']:
            return ('Y', a, LEFT)
        if a == ' ':
            return ('accept', ' ', RIGHT)
    return ('reject', a, STAY)


M = TuringMachine('X', trans)
M.compute('0010')
runtheTM(M)
```



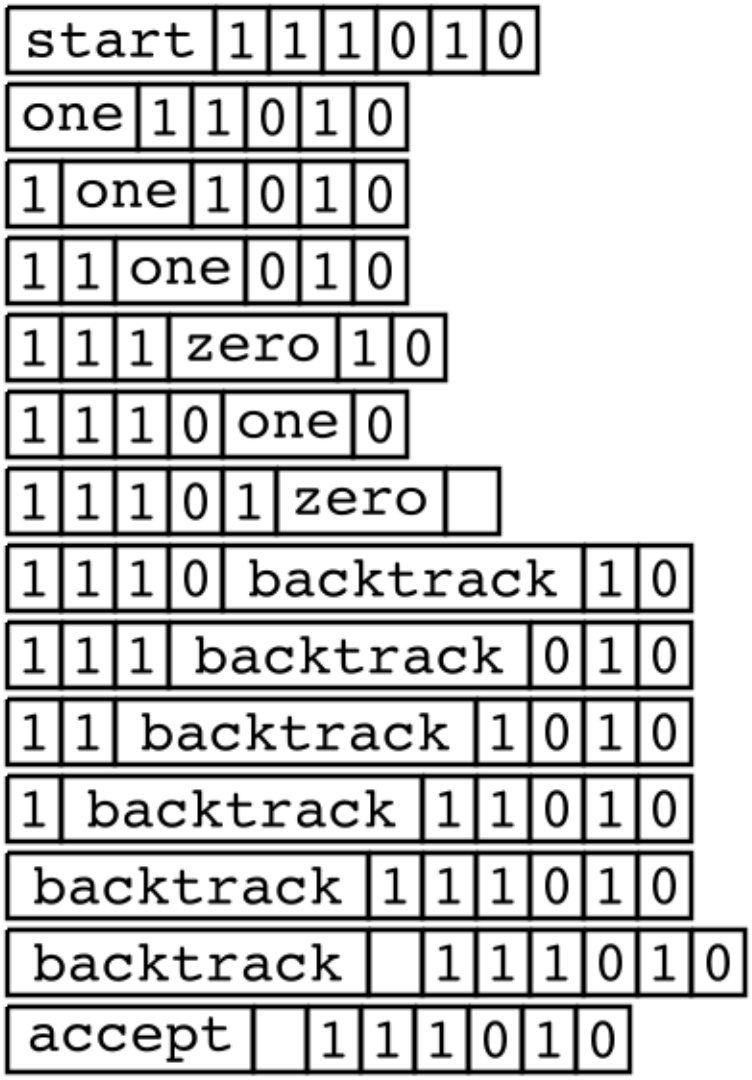
The transition function `trans` has a catchall return statement at the end to reject anything other states or symbols.

```
M = TuringMachine('X', trans)
M.compute('0010000002')
runtheTM(M)
```



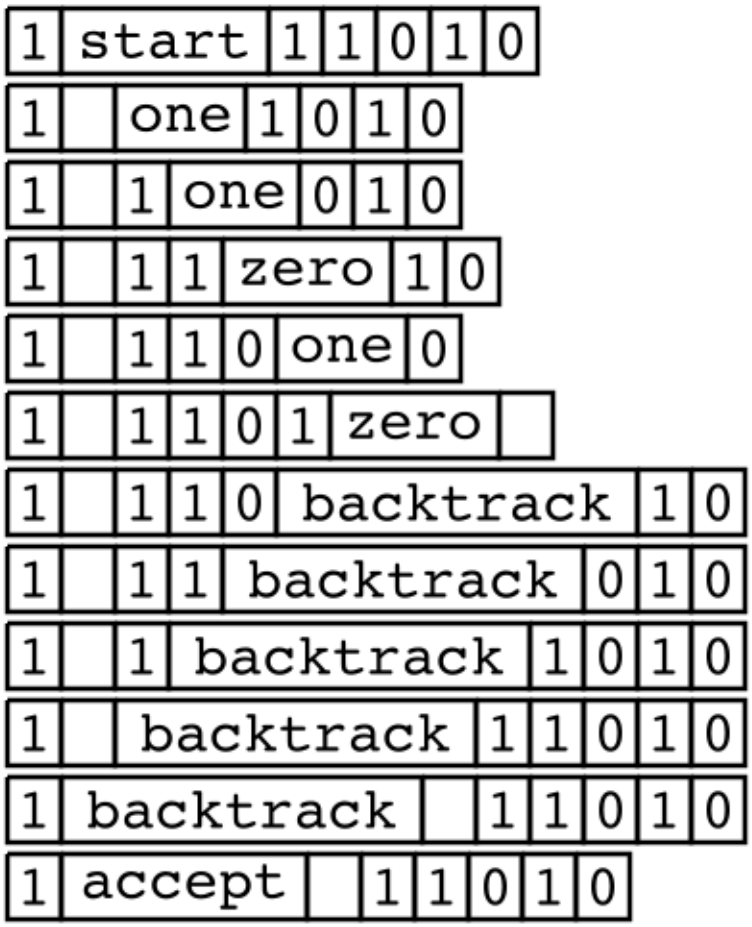
Let's implement the moveover functionality described in class. Recall that this TM should put a blank space at the head position and shift all the other tape cells to the right by one step.

```
def moveover(q, a):
    if q == 'start':
        nextstate = 'zero' if a == '0' else 'one'
        return (nextstate, ' ', RIGHT)
    if q in ['zero', 'one']:
        symbol_to_write = '0' if q == 'zero' else '1'
        if a == ' ':
            return ('backtrack', symbol_to_write, LEFT)
        elif a == '0':
            return ('zero', symbol_to_write, RIGHT)
        else:
            return ('one', symbol_to_write, RIGHT)

    if q == 'backtrack':
        if a in ['0', '1']:
            return ('backtrack', a, LEFT)
        else:
            return ('accept', ' ', STAY)
    return ('reject', None, STAY)



M = TuringMachine('start', moveover)
M.compute('111010')
runtheTM(M)
```



This is a little unsatisfying, because we could have achieved the same result by simply moving left at the first step. The power of this little machine shows up when we are not at the beginning of the tape. We can artificially move the tape head to another position before we run it and we will see the results is more interesting.

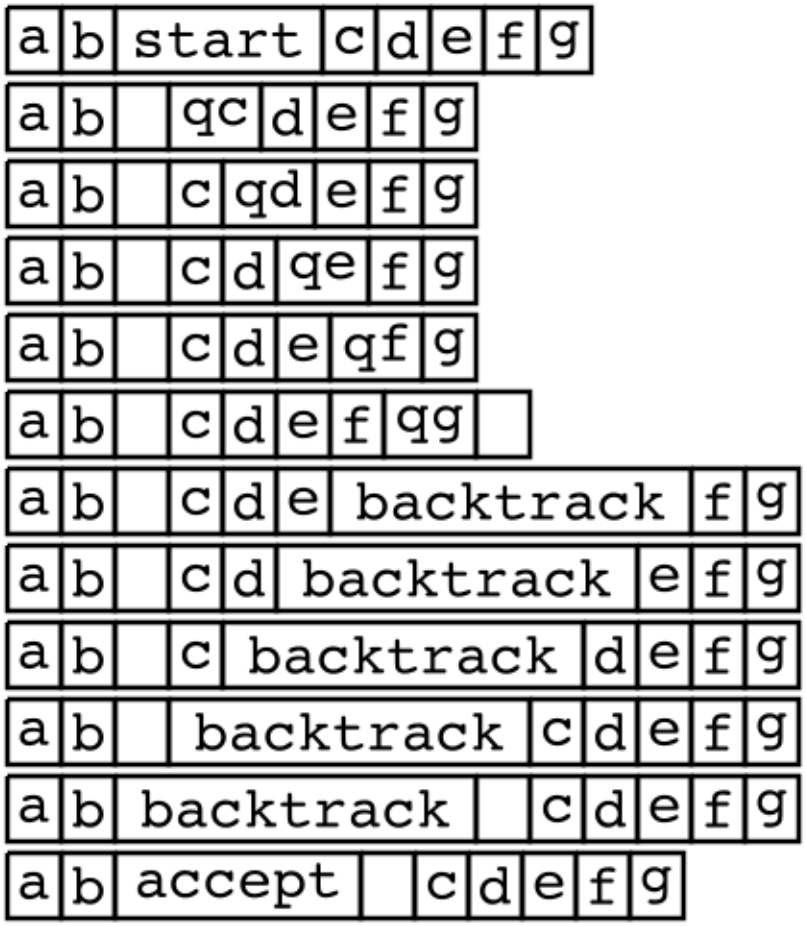
```
M = TuringMachine('start', moveover)
M.compute('111010')
M.tape.move(RIGHT)
runtheTM(M)
```



It would be nice if this worked for other alphabets. The natural thing to do here is to have a state for each symbol. For a symbol `'x'`, we will make a state `'qx'`. That is if the symbol is stored in a string `a`, we will make a state `'q' + a`. Then, when in state `q = 'qx'`, we can access the symbol to write as `q[1]`. Here it is in code.

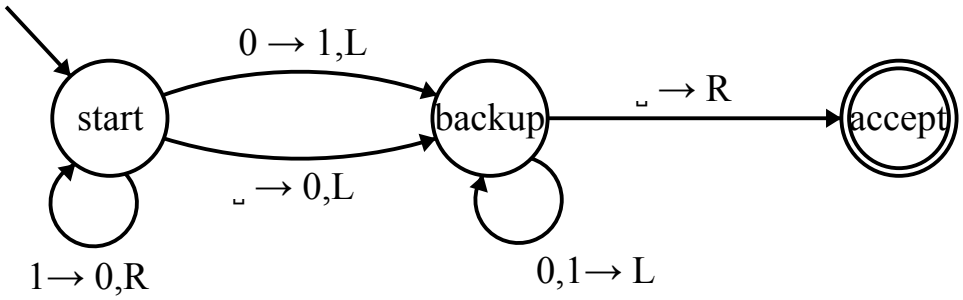
```
def moveover(q, a):
    if q == 'start':
        return ('q' + a, ' ', RIGHT)
    elif q == 'backtrack':
        if a == ' ':
            return ('accept', ' ', STAY)
        else:
            return ('backtrack', a, LEFT)
    else:
        if a == ' ':
            return ('backtrack', q[1], LEFT)
        else:
            return ('q'+ a, q[1], RIGHT)
```

```
M = TuringMachine('start', moveover)
M.compute('abcdefg')
M.tape.move(RIGHT)
M.tape.move(RIGHT)
runtheTM(M)
```



Enumerating Binary Strings

Suppose we want to enumerate all binary strings. (In fact, we do want to do this in order to simulate nondeterministic TMs.) One way to do this is to make a Turing Machine that will start with a binary string on the tape and will update the tape so that it has the lexicographically next binary string. It's a little easier if we imagine that the strings are written backwards on the tape. This is like treating the tape as a number in binary with the least significant bit starting on the left and then adding one. The one difference is that after 111111 comes 00000000. More generally, we count up to pp copies of 11 and then replace it with $p+1$p + 1 copies of 0 0. Here is a TM that implements this process.




```
def binary(state, bit):
    if state == 'start':
        if bit == '0':
            return ('backup', '1', LEFT)
        if bit == '1':
            return ('start', '0', RIGHT)
        if bit == ' ':
            return ('backup', 0, LEFT)
    elif state == 'backup':
        if bit in {'0', '1'}:
            return ('backup', bit, LEFT)
        if bit == ' ':
            return ('accept', ' ', RIGHT)
    else:
        return ('reject', bit, STAY)


M = TuringMachine('start', binary)
M.compute('11100001')
runtheTM(M)
```


```
M = TuringMachine('start', binary)
M.compute('11111')
runtheTM(M)
```

A more complicated state

We can think of the state as being decomposed into variables.

Let's design a TM that recognizes the following language.

$A = \{w\#w \mid w \in \{0,1\}^*\}$ A = {w#w ∣ w ∈ {0, 1}∗}

Here is an informal description of the machine. The state for such a machine will include both what step of the program we are in as well as the values of the variables. In this case, there is just one variable called `m` that stores a symbol. Working with variables this way really requires us to be careful that there are only a finite set of values that the variable may take.

1. If the current symbol is `#` go to step 7. Otherwise, set `m` to be the current symbol, write an `x` to the tape, and go right.
2. Move right until you reach a `#` symbol. Reject if you hit a space.

3. Skip over any x's.
4. Check that the current symbol matches m and write an `x`. Otherwise, reject.
5. Move left until you reach a `#` symbol.
6. Move left until you reach an `x`, then move right and go to 1.
7. Move right until you reach a `' '` and accept. Reject if you see anything other than an `x` along the way.

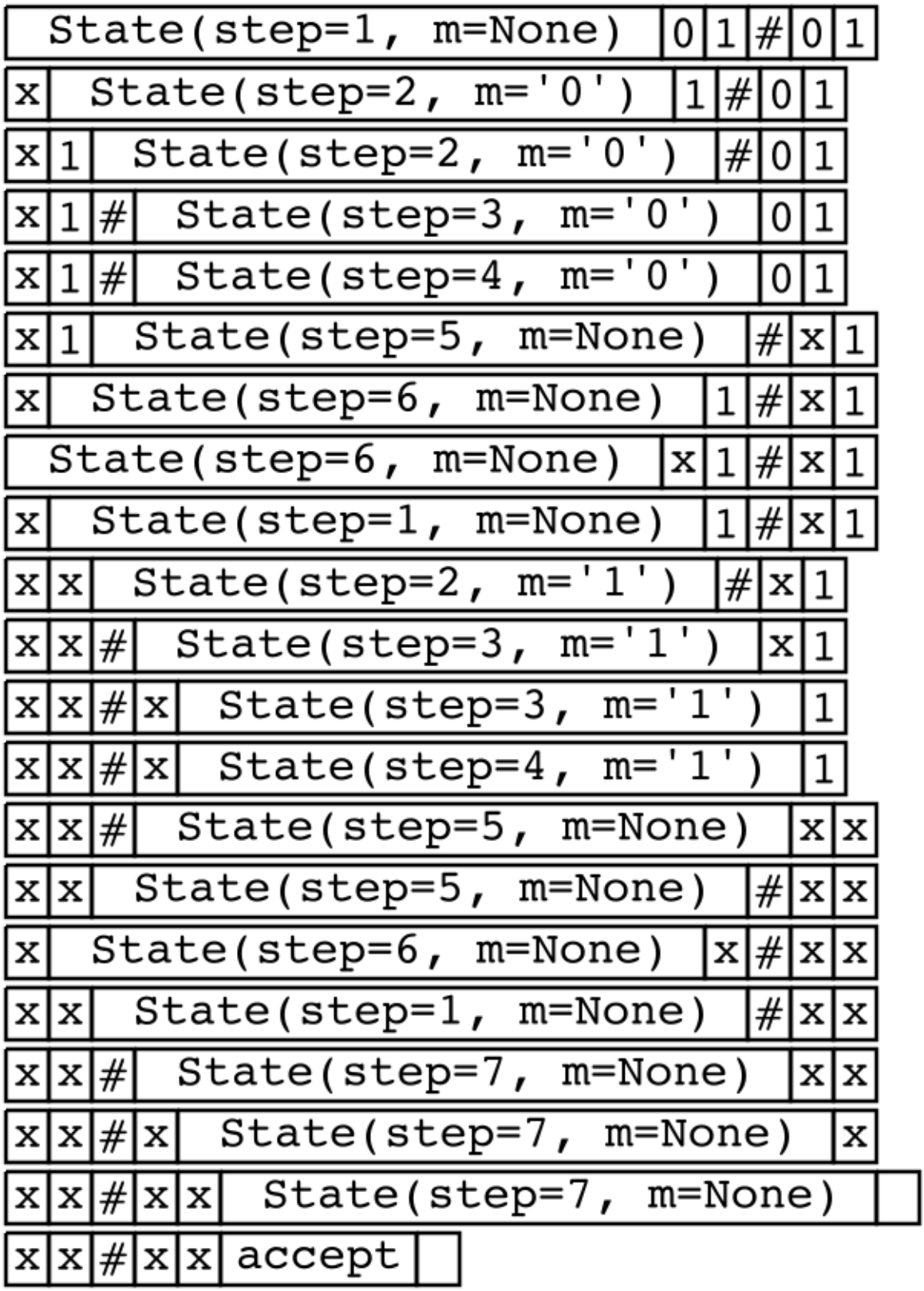
Now we can convert this informal description into a transition function. In this case, it is easier to write the transition function than it would be to draw out the state diagram.

```
from collections import namedtuple

State = namedtuple('State', ['step', 'm'])

start = State(step = 1, m = None)
reject = ('reject', None, STAY)
accept = ('accept', None, STAY)

def duplicate(state, symbol):
    if state.step == 1:
        if symbol == '#':
            return (State(step = 7, m = None), None, RIGHT)
        else:
            return (State(step = 2, m = symbol), 'x', RIGHT)
    elif state.step == 2:
        if symbol == '#':
            return (State(step = 3, m = state.m), None, RIGHT)
        elif symbol == ' ':
            return reject
        else:
            return (state , None, RIGHT)
    elif state.step == 3:
        if symbol == 'x':
            return (state, None, RIGHT)
        else:
            return (State(step = 4, m = state.m), symbol, STAY)
    elif state.step == 4:
        if symbol == state.m:
            return (State(step = 5, m = None), 'x', LEFT)
        else:
            return reject
    elif state.step == 5:
        if symbol == '#':
            return (State(step = 6, m = None), None, LEFT)
        else:
            return (state, None, LEFT)
    elif state.step == 6:
        if symbol == 'x':
            return (start, None, RIGHT)
        else:
            return (state, None, LEFT)
    elif state.step == 7:
        if symbol == 'x':
            return (state, None, RIGHT)
        elif symbol == ' ':
            return accept
        else:
            return reject

M = TuringMachine(start, duplicate)
```

```
M.compute('0#01')
runtheTM(M)
```

State(step=1, m=None)	0	#	0	1
x	State(step=2, m='0')	#	0	1
x	#	State(step=3, m='0')	0	1
x	#	State(step=4, m='0')	0	1
x	State(step=5, m=None)	#	x	1
State(step=6, m=None)	x	#	x	1
x	State(step=1, m=None)	#	x	1
x	#	State(step=7, m=None)	x	1
x	#	x	State(step=7, m=None)	1
x	#	x	reject	1

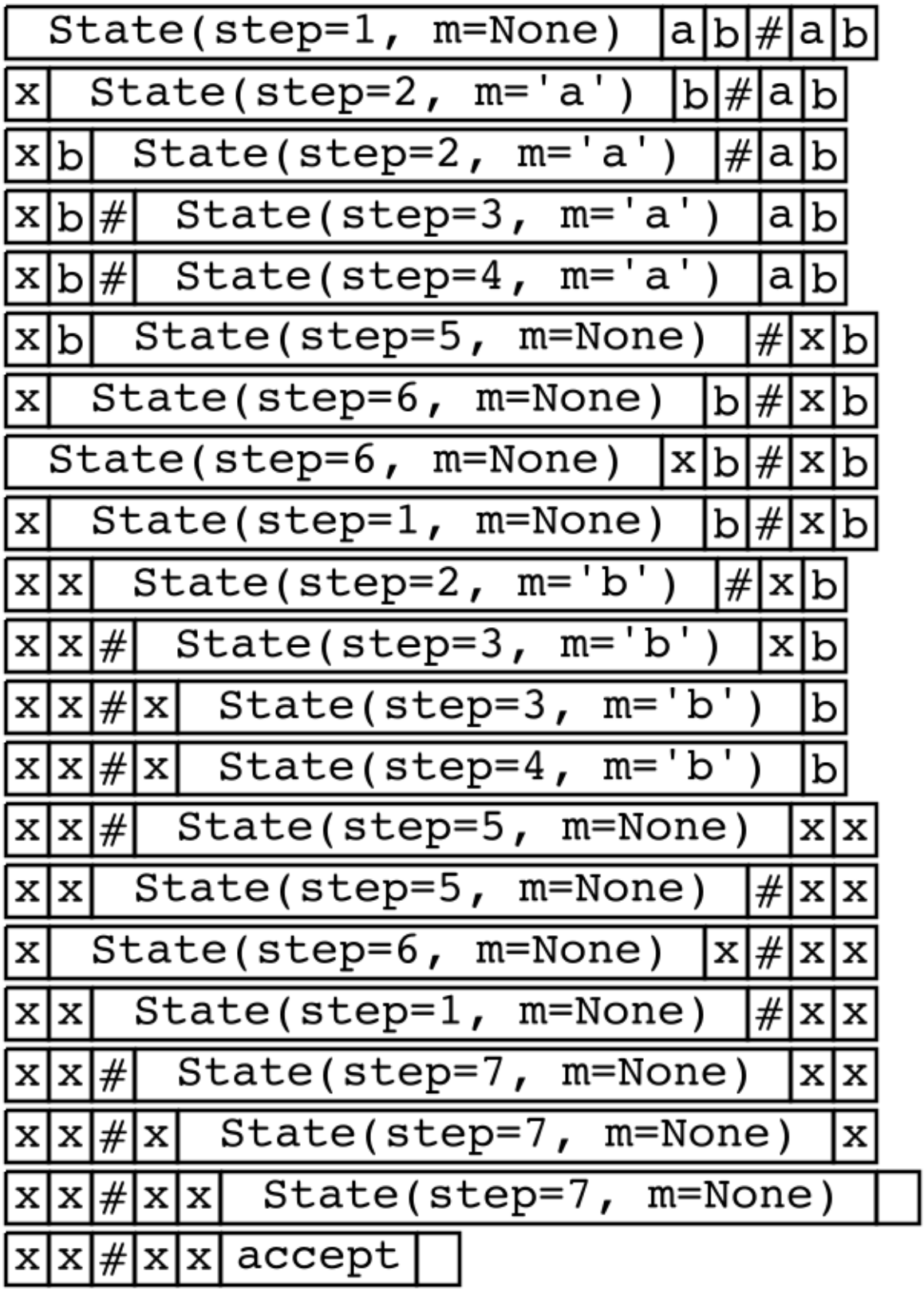
```
M.compute('01#01')
runtheTM(M)
```



It turns out that our code above works for other alphabets as well. If we were to have an alphabet with 10 symbols, we would need to have $7 \times 10 + 2 = 727 \times 10 + 2 = 72$ states. That's just too many to draw. We could hope that many of the states are not reachable, but there is no way to avoid that the number of states will grow linearly with the number of symbols. This is at least some justification for thinking seriously about the Turing

Machine in terms of its transition function first, and as a state diagram second. Remember that the state diagram only gives us a visual representation of the transition function.

```
M.compute('ab#ab')
runtheTM(M)
```



Here is the sequence of tapes for a longer example.

```
M.compute('0000#0000')
print(M.tape, M.state)
while M.state not in {'accept', 'reject'}:
    M.step()
    print(M.tape, M.state)
```

```
0000#0000 State(step=1, m=None)
x000#0000 State(step=2, m='0')
x000#0000 State(step=2, m='0')
x000#0000 State(step=2, m='0')
x000#0000 State(step=2, m='0')
x000#0000 State(step=3, m='0')
x000#0000 State(step=4, m='0')
x000#x000 State(step=5, m=None)
x000#x000 State(step=6, m=None)
x000#x000 State(step=6, m=None)
x000#x000 State(step=6, m=None)
x000#x000 State(step=6, m=None)
x000#x000 State(step=1, m=None)
xx00#x000 State(step=2, m='0')
xx00#x000 State(step=2, m='0')
xx00#x000 State(step=2, m='0')
xx00#x000 State(step=3, m='0')
xx00#x000 State(step=3, m='0')
xx00#x000 State(step=4, m='0')
xx00#xx00 State(step=5, m=None)
xx00#xx00 State(step=5, m=None)
xx00#xx00 State(step=6, m=None)
xx00#xx00 State(step=6, m=None)
xx00#xx00 State(step=6, m=None)
xx00#xx00 State(step=1, m=None)
xxx0#xx00 State(step=2, m='0')
xxx0#xx00 State(step=2, m='0')
xxx0#xx00 State(step=3, m='0')
xxx0#xx00 State(step=3, m='0')
xxx0#xx00 State(step=3, m='0')
xxx0#xx00 State(step=4, m='0')
xxx0#xxx0 State(step=5, m=None)
xxx0#xxx0 State(step=5, m=None)
xxx0#xxx0 State(step=5, m=None)
xxx0#xxx0 State(step=6, m=None)
xxx0#xxx0 State(step=6, m=None)
xxx0#xxx0 State(step=1, m=None)
xxxx#xxx0 State(step=2, m='0')
xxxx#xxx0 State(step=3, m='0')
xxxx#xxx0 State(step=3, m='0')
xxxx#xxx0 State(step=3, m='0')
xxxx#xxx0 State(step=3, m='0')
xxxx#xxx0 State(step=4, m='0')
xxxx#xxxx State(step=5, m=None)
xxxx#xxxx State(step=5, m=None)
```

```
xxxx#xxxx State(step=5, m=None)
xxxx#xxxx State(step=5, m=None)
xxxx#xxxx State(step=6, m=None)
xxxx#xxxx State(step=1, m=None)
xxxx#xxxx State(step=7, m=None)
xxxx#xxxx State(step=7, m=None)
xxxx#xxxx State(step=7, m=None)
xxxx#xxxx State(step=7, m=None)
xxxx#xxxx  State(step=7, m=None)
xxxx#xxxx  accept
```