

## Task 1: Illustrating Decidability of a Computational Problem

For this task I have chosen to illustrate the decidability of  $A_{DFA} = \{\langle B, w \rangle \mid B \text{ is a DFA that accepts string } w\}$ .

### String Representation

For this problem I have decided to represent  $\langle B, w \rangle = x$  where  $B$  is a DFA and  $w$  is a string over  $B$ 's alphabet as a JSON string with the following format:

```
1 {  
    states: string[]  
3    alphabet: char[]  
    transitions: tuple<string, string, string>[]  
5    start_state: string  
    accept_states: string[]  
7    word: string  
}
```

**states** is an array of states in the DFA

**alphabet** is an array of characters in the DFA's alphabet

**transitions** is an array of 3-tuples defining the DFA's transition function where the first string is the current state, the second string is the character being read, and the third string is the output state.

**start\_state** is the start state of the DFA

**accept\_states** is an array representing the set of acceptance states in the DFA

**word** is  $w$ , the string whose membership in  $L(B)$  is being tested

### Parsing and Computation

The input string  $x$  is parsed by first reading the JSON string into a dictionary.

Then the program ensures that  $x$  accurately represents  $\langle B, w \rangle$  by making sure the **start\_state** is a member of **states**, **accept\_states** is a subset of **states**, **word** is a string over **alphabet**, and **transitions** defines the proper transitions such that all states have *exactly one* transition defined for every member of the alphabet and the output of the transition function is always a member of **states**, aka  $\delta : Q \times \Sigma \rightarrow Q$ .

After the input string  $x$  is validated, the arrays are transformed to sets and the transition function is represented as a adjacency-list like dictionary.

If any errors are thrown in the parsing and validation of  $x$  the program halts and returns **false** because that means  $x \neq \langle B, w \rangle$ .

If the input  $x$  is valid, the program then simulates the computation of  $B$  on  $w$  and if the computation ends in an accept state the program returns **true**, otherwise it returns **false**.

### Example Strings

For both of the following examples, the input string represents the following DFA from example 1.9 in the textbook where  $L(B) = \{w \mid w \text{ is the empty string or ends in } 0\}$ .  $B$  has the following state diagram:

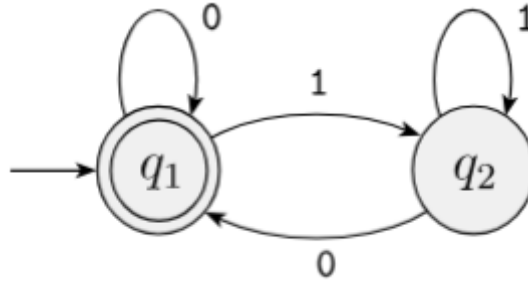


Figure 1: Textbook Example 1.9

### Accepted String

For an accepted string let  $x = \langle B, w \rangle$  where  $B$  is the DFA shown in Figure 1 and  $w = 100$ . Here is  $x$ :

```

{
2   "states": ["q1", "q2"],
   "alphabet": ["0", "1"],
4   "transitions": [
       ["q1", "0", "q1"],
       ["q1", "1", "q2"],
       ["q2", "1", "q2"],
       ["q2", "0", "q1"]
   ],
10  "start_state": "q1",
   "accept_states": ["q1"],
12  "word": "100"
}

```

$x \in A_{DFA}$  because it properly encodes  $\langle B, w \rangle$  and  $w \in L(B)$  because 100 ends in 0. The computation of  $B$  on  $w$  has the following state transitions:  $q1 \rightarrow q2 \rightarrow q1 \rightarrow q1$  and since  $q1$  is an accept state  $B$  accepts  $w$ .

When running the program on this input  $x$ , we see the program correctly returns **true**. Note that instead of typing the long JSON string manually, I have it saved to a file and am passing it in as a command line arg with `cat`.

```

[michael@nixos:~/School/15th_Grade/CSE105/project]$ elixir lib/dfa.exs "$(cat dfa_specs/ends_in_zero_accept.json)"
true
[michael@nixos:~/School/15th_Grade/CSE105/project]$

```

Figure 2: Running the program on  $x \in A_{DFA}$

### Rejected String

For a rejected string let  $x = \langle B, w \rangle$  where  $B$  is the DFA shown in Figure 1 and  $w = 00001$ . Here is  $x$ :

```

{
2   "states": ["q1", "q2"],
   "alphabet": ["0", "1"],
4   "transitions": [
       ["q1", "0", "q1"],

```

```

6      ["q1", "1", "q2"],
      ["q2", "1", "q2"],
8      ["q2", "0", "q1"]
    ],
10    "start_state": "q1",
    "accept_states": ["q1"],
12    "word": "00001"
  }

```

$x \notin A_{DFA}$  because although it properly encodes  $\langle B, w \rangle$ ,  $w \notin L(B)$  because it ends in a 1. The computation of  $B$  on  $w$  has the following state transitions:  $q1 \rightarrow q1 \rightarrow q1 \rightarrow q1 \rightarrow q1 \rightarrow q2$  and since  $q2$  is not an accepting state,  $B$  rejects  $x$ .

When running the program on this input  $x$ , we see the program correctly returns **false**. Note that instead of typing the long JSON string manually, I have it saved to a file and am passing it in as a command line arg with `cat`.

```

[michael@nixos:~/School/15th_Grade/CSE105/project]$ elixir lib/dfa.exs "$(cat dfa_specs/ends_in_zero_reject.json)"
false
[michael@nixos:~/School/15th_Grade/CSE105/project]$

```

Figure 3: Running the program on  $x \notin A_{DFA}$

## Video Explanation

Watch the video [here](#)

## Code

If you'd like to read the code in a better viewing format than PDF, visit the file on [GitHub](#)

```

defmodule Project.DFA do
2  @doc """
    Parses a JSON string representing  $\langle B, w \rangle$  where  $B$  is any DFA and  $w$ 
4  is a string over the alphabet of  $B$  to run the computation on. From the
    string this
    validates  $M$  and gets the formal definition of states,
6  alphabet, transition function, start state, and accept states while
    also validating the string  $w$  to run the computation on is valid
8
    ## JSON DFA Format
10  {
    states: string[]
12  alphabet: char[]
    transitions: tuple<string, string, string>[]
14  start_state: string
    accept_states: string[]
16  word: string
    }
18  states is an array of states in the DFA
    alphabet is an array of characters in the DFA's alphabet
20  transitions is an array of 3-tuples defining
    the DFA's transition function where the first string is the

```

```

22  current state, the second string is the character being read,
    and the third string is the output state
24  start_state is the start state of the DFA
    accept_states is an array representing the set of acceptance states in
        the DFA.
26  word is the string whose membership in the DFA will be tested

28  Returns the DFA with the arrays cast to sets and the transitions as an
    adjacency list as well as the input string w
30  """
    def parse_dfa(input) do
32      %{
        "states" => states,
34      "alphabet" => alphabet,
        "transitions" => transitions,
36      "start_state" => start_state,
        "accept_states" => accept_states,
38      "word" => word
    } = :json.decode(input)

40
    states = MapSet.new(states)
42    alphabet = MapSet.new(alphabet)
    accept_states = MapSet.new(accept_states)
44    # map the transition 3 tuples into an adjacency list
    transitions = Enum.reduce(transitions, %{}, fn [start, label, dest],
46      acc ->
        Map.update(
48          acc,
          start,
          %{label => dest},
50          fn existing ->
            # if this state already has an existing transition with the
              same label (input char)
52            # raise an error
            if Map.has_key?(existing, label) do
54              raise "A state cannot have two transitions with the same
                label"
            end
56            Map.put(existing, label, dest)
          end
58          )
        end)

60
    # assert that all accept states are members of states, aka
      accept_states subseteq states
62    if not MapSet.subset?(accept_states, states) do
      raise "All accept states must be members of the states array"
64    end

    # assert that the start state is a member of states
    if not MapSet.member?(states, start_state) do
66      raise "The start state must be a member of the state array"
68    end
70

```

```

72   # assert that every state has transition rules
    if not MapSet.equal?(states, transitions |> Map.keys |> MapSet.new) do
      raise "Every state must have transitions defined"
74   end
    # assert that every state has a transition for every member of the
      alphabet and said transition maps to a valid state
76   transitions
      |> Map.values
78   |> Enum.each(fn rules ->
      # assert that the state's rules has a rule for each alphabet
        character
80     if not MapSet.equal?(alphabet, rules |> Map.keys |> MapSet.new) do
      raise "There must be a transition defined for every alphabet
        character"
82     end
      # assert that every destination in the rules is a valid state
84
      if not MapSet.subset?(rules |> Map.values |> MapSet.new, states)
      do
86       raise "Every destination state must be a member of the state
        array"
      end
88     end)

90   # now the DFA input is guaranteed to be valid so assert that the
      input string
      # is valid
92   word = String.graphemes(word)
      # assert that the input string only contains characters in the
        alphabet
94   if not MapSet.subset?(word |> MapSet.new, alphabet) do
      raise "word must only contain characters defined in the alphabet
        list"
96   end
      {states, alphabet, transitions, start_state, accept_states, word}
98   end

100  @doc """
    Given the encoded <B, w>, decides whether w is in L(B)
102
    Assuming the input string is valid, the computation of it goes as
      follows:
104   Start with the current state at start_state. Then, read the string
      right to left.
      For every character, modify the current state to be the result of the
        transition
106   function with the current state and current character being read.
      After the string is processed, check if the current state is an accept
        state. If
108   it is, accept, otherwise, reject.
      ""
110   def test_decidable(input) do
      {_, _, transitions, start_state, accept_states, word} =
        parse_dfa(input)

```

```

112     # process input string
114     end_state = Enum.reduce(word, start_state, fn character, state ->
        Map.get(transitions, state) |> Map.get(character)
116     end)
        MapSet.member?(accept_states, end_state)
118 end

120 def call do
    # if an error is thrown during the testing, return false (failed the
        typecheck)
122     try do
        test_decidable(System.argv() |> Enum.at(0, ""))
124     rescue
        _ -> false
126     end
    end
128 end

130 Project.DFA.call() |> IO.puts

```

## Task 2: Illustrating a Mapping Reduction

For this task I have chosen to illustrate the mapping reduction  $A_{TM} \leq_m HALT_{TM}$  where

$$A_{TM} = \{\langle M, w \rangle \mid M \text{ is a Turing machine, } w \text{ is a string, and } w \in L(M)\}$$

$$HALT_{TM} = \{\langle M, w \rangle \mid M \text{ is a Turing machine, } w \text{ is a string, and } M \text{ halts on } w\}$$

As defined in lecture both  $A_{TM}$  and  $HALT_{TM}$  are undecidable and  $A_{TM} \leq_m HALT_{TM}$  can be done with the function  $F : \Sigma^* \rightarrow \Sigma^*$  defined as follows

$$F = \begin{cases} const_{out} & \text{if } x \neq \langle M, w \rangle \text{ for any Turing machine } M \text{ and string } w \text{ over the alphabet of } M \\ \langle M', w \rangle & \text{if } x = \langle M, w \rangle \text{ for some Turing machine } M \text{ and string } w \text{ over the alphabet of } M \end{cases}$$

where  $const_{out}$  is some constant  $\langle M, w \rangle \notin HALT_{TM}$  and  $M'$  is a Turing machine that computes like  $M$  except if the computation of  $M$  were ever to go to a reject state,  $M'$  loops instead.

## String Representation

Similar to task 1,  $\langle M, w \rangle$  is encoded as a JSON string with the following format:

```

{
2   states: string[]
   input_alphabet: char[]
4   tape_alphabet: char[]
   transitions: tuple<string, string, string, string, string>[]
6   start_state: string
   accept_state: string
8   reject_state: string
   word: string
10 }

```

`states` is an array of states in the Turing machine

`input_alphabet` is the array of characters in the Turing machine's input alphabet

`tape_alphabet` is the array of characters in the Turing machine's tape alphabet.

`transitions` is an array of 5 tuples specifying the Turing machine's transition function where the first string is the input state, the second string is the character being read, the third string is the character to write, the fourth string is the direction to move the tape head (either 'R' or 'L'), and the fifth string is the output state.

`start_state` is the Turing machine's starting state.

`accept_state` is the Turing machine's accept state.

`reject_state` is the Turing machine's reject state.

`word` is the input  $w$ , a string over the `input_alphabet`

## Parsing

Given the input string  $x = \langle M, w \rangle$ , the string is first parsed as JSON into a dictionary.

The program then checks that  $x = \langle M, w \rangle$  by making sure `start_state` is a member of `states`, `input_alphabet` is a subset of `tape_alphabet`, `accept_state` is a member of `states`, `reject_state` is a member of `states`, `word` is the string  $w$  which is over `input_alphabet` and `transitions` properly defines the transition function such that all states have *exactly one* transition defined for every member of `tape_alphabet` and the transition is valid such that the character being read/written is in the tape alphabet, the tape head is being moved either right or left, and the output state is a member of `states`, aka  $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ .

After the input string  $x$  is validated, the arrays are transformed to sets and the transition function is represented as a adjacency-list like dictionary.

If any errors are thrown in the parsing and validation of  $x$  the program halts and returns `constout`.

For this program, `constout` =  $\langle A, \epsilon \rangle$  where  $A$  has the following state diagram:

$A$  never halts because it loops on its starting state, which means that  $\langle A, \epsilon \rangle \notin HALT_{TM}$ , therefore it is an appropriate value for `constout`.

## Mapping

Assuming parsing went well, the input is then mapped from  $\langle M, w \rangle \in A_{TM}$  to  $\langle M', w \rangle \in HALT_{TM}$  with  $F$ .

A new state `loop_state` is added to the states and transition function is modified such that on `loop_state` the computation will loop forever by defining the transition rules such that no matter what symbol is read, the Turing machine will leave the tape unchanged, move the tape head right and stay on `loop_state`.

Then, for every transition in the transition function that goes to `reject_state`, the transition is changed to go to `loop_state` instead.

This process produces  $M'$  from  $M$  and since  $w$  is unchanged by  $F$ , the program can then serialize  $\langle M', w \rangle$  into the same JSON format as the input, output that string, and halt.

## Examples

For both examples I'll use the following Turing machine  $M$  from the textbook figure 3.8.

This Turing machine  $M$  decides  $A = \{0^{2^n} | n \geq 0\}$ .

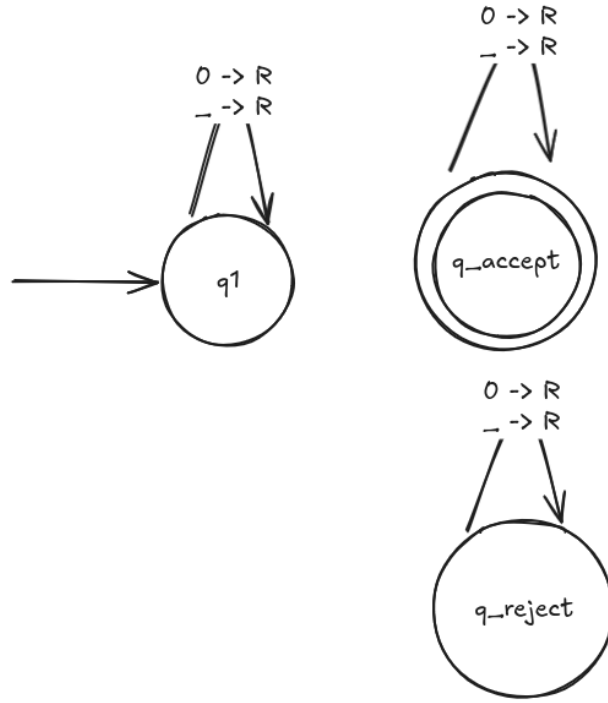


Figure 4: A's State Diagram

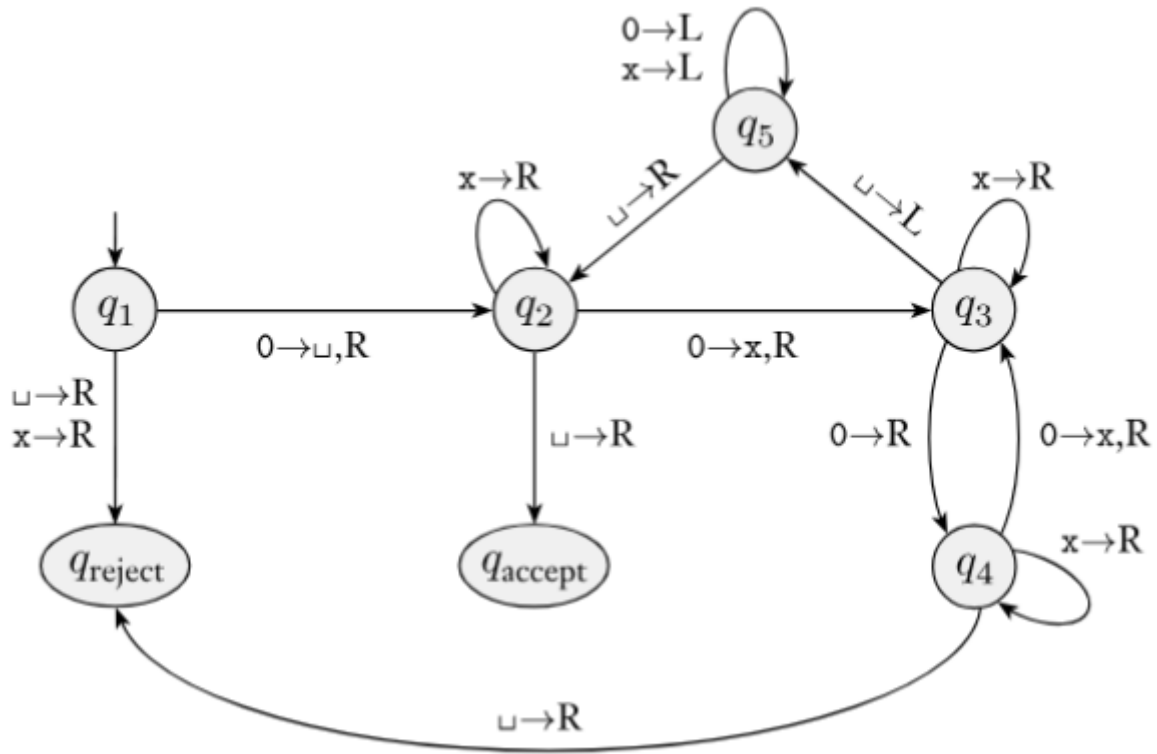


Figure 5: Textbook Figure 3.8



### Positive Instance

For a positive instance of a string  $x = \langle M, w \rangle$  where  $M$  is a Turing machine and  $w$  is a string over the Turing machine's input alphabet, let's use the Turing machine defined in figure 5.

This Turing machine  $M$  decides  $A = \{0^{2^n} | n \geq 0\}$  and the string  $w$  I have chosen is  $w = 0000 = 0^{2^2}$  such that  $\langle M, w \rangle \in A_{TM}$  because  $w \in L(A)$ . To show  $w \in L(A)$ , the computation of  $w$  on  $M$  goes as follows (read pointer is 0 indexed):

Tape: 0000\_\_\_\_, State:  $q_1$ , Read pointer in position 0  
Tape: \_000\_\_\_\_, State:  $q_2$ , Read pointer in position 1  
Tape: \_x00\_\_\_\_, State:  $q_3$ , Read pointer in position 2  
Tape: \_x00\_\_\_\_, State:  $q_4$ , Read pointer in position 3  
Tape: \_x0x\_\_\_\_, State:  $q_3$ , Read pointer in position 4  
Tape: \_x0x\_\_\_\_, State:  $q_5$ , Read pointer in position 3  
Tape: \_x0x\_\_\_\_, State:  $q_5$ , Read pointer in position 2  
Tape: \_x0x\_\_\_\_, State:  $q_5$ , Read pointer in position 1  
Tape: \_x0x\_\_\_\_, State:  $q_5$ , Read pointer in position 0  
Tape: \_x0x\_\_\_\_, State:  $q_2$ , Read pointer in position 1  
Tape: \_x0x\_\_\_\_, State:  $q_2$ , Read pointer in position 2  
Tape: \_xxx\_\_\_\_, State:  $q_3$ , Read pointer in position 3  
Tape: \_xxx\_\_\_\_, State:  $q_3$ , Read pointer in position 4  
Tape: \_xxx\_\_\_\_, State:  $q_5$ , Read pointer in position 3  
Tape: \_xxx\_\_\_\_, State:  $q_5$ , Read pointer in position 2  
Tape: \_xxx\_\_\_\_, State:  $q_5$ , Read pointer in position 1  
Tape: \_xxx\_\_\_\_, State:  $q_5$ , Read pointer in position 0  
Tape: \_xxx\_\_\_\_, State:  $q_2$ , Read pointer in position 1  
Tape: \_xxx\_\_\_\_, State:  $q_2$ , Read pointer in position 2  
Tape: \_xxx\_\_\_\_, State:  $q_2$ , Read pointer in position 3  
Tape: \_xxx\_\_\_\_, State:  $q_2$ , Read pointer in position 4  
Tape: \_xxx\_\_\_\_, State:  $q_{accept}$ , Read pointer in position 5

The computation halts in  $q_{accept}$ , therefore  $w \in L(M)$  and  $\langle M, w \rangle \in A_{TM}$ .

Given this  $\langle M, w \rangle$ ,  $x$  is the following JSON string:

```
{
  "states": ["q1", "q2", "q3", "q4", "q5", "q_accept", "q_reject"],
  "input_alphabet": ["0"],
  "tape_alphabet": ["0", "x", "_"],
  "transitions": [
    ["q1", "_", "_", "R", "q_reject"],
    ["q1", "x", "x", "R", "q_reject"],
    ["q1", "0", "_", "R", "q2"],
    ["q2", "_", "_", "R", "q_accept"],
```

```

12      ["q2", "x", "x", "R", "q2"],
      ["q2", "0", "x", "R", "q3"],

14      ["q3", "_", "_", "L", "q5"],
      ["q3", "x", "x", "R", "q3"],
16      ["q3", "0", "0", "R", "q4"],

18      ["q4", "_", "_", "R", "q_reject"],
      ["q4", "x", "x", "R", "q4"],
20      ["q4", "0", "x", "R", "q3"],

22      ["q5", "_", "_", "R", "q2"],
      ["q5", "x", "x", "L", "q5"],
24      ["q5", "0", "0", "L", "q5"],

26      ["q_accept", "_", "_", "R", "q_accept"],
      ["q_accept", "x", "x", "R", "q_accept"],
28      ["q_accept", "0", "0", "R", "q_accept"],

30      ["q_reject", "_", "_", "R", "q_reject"],
      ["q_reject", "x", "x", "R", "q_reject"],
32      ["q_reject", "0", "0", "R", "q_reject"]
    ],
34    "start_state": "q1",
      "accept_state": "q_accept",
36    "reject_state": "q_reject",
      "word": "0000"
38  }

```

This is a positive input to the mapping reduction function because  $x = \langle M, w \rangle$  where  $M$  is a well-defined Turing machine and  $w$  is a string over  $M$ 's input alphabet and  $\langle M, w \rangle \in A_{TM}$ . Therefore this should result in the JSON string  $\langle M', w \rangle \in HALT_{TM}$  where  $w$  is unchanged from the input and  $M'$  is a Turing machine that computes like  $M$  except if the computation of  $M$  were ever to go to a reject state,  $M'$  loops instead.  $M'$  should have the following state diagram:

$\langle M', w \rangle \in HALT_{TM}$  because running the computation of  $M'$  on  $w$  results in the following computations:

Tape: 0000\_\_\_\_, State:  $q_1$ , Read pointer in position 0

Tape: \_000\_\_\_\_, State:  $q_2$ , Read pointer in position 1

Tape: \_x00\_\_\_\_, State:  $q_3$ , Read pointer in position 2

Tape: \_x00\_\_\_\_, State:  $q_4$ , Read pointer in position 3

Tape: \_x0x\_\_\_\_, State:  $q_3$ , Read pointer in position 4

Tape: \_x0x\_\_\_\_, State:  $q_5$ , Read pointer in position 3

Tape: \_x0x\_\_\_\_, State:  $q_5$ , Read pointer in position 2

Tape: \_x0x\_\_\_\_, State:  $q_5$ , Read pointer in position 1

Tape: \_x0x\_\_\_\_, State:  $q_5$ , Read pointer in position 0

Tape: \_x0x\_\_\_\_, State:  $q_2$ , Read pointer in position 1

Tape: \_x0x\_\_\_\_, State:  $q_2$ , Read pointer in position 2

Tape: \_xxx\_\_\_\_, State:  $q_3$ , Read pointer in position 3



```

2   "accept_state":"q_accept",
   "input_alphabet":[
4     "0"
   ],
6   "reject_state":"q_reject",
   "start_state":"q1",
8   "states":[
   "q1", "q2", "q3", "q4", "q5", "q_accept", "q_loop_grgat", "q_reject"
10  ],
   "tape_alphabet":["0", "_", "x"],
12  "transitions":[
   ["q1", "0", "_", "R", "q2"],
14  ["q1", "_", "_", "R", "q_loop_grgat"],
   ["q1", "x", "x", "R", "q_loop_grgat"],
16  ["q2", "0", "x", "R", "q3"],
   ["q2", "_", "_", "R", "q_accept"],
18  ["q2", "x", "x", "R", "q2"],
   ["q3", "0", "0", "R", "q4"],
20  ["q3", "_", "_", "L", "q5"],
   ["q3", "x", "x", "R", "q3"],
22  ["q4", "0", "x", "R", "q3"],
   ["q4", "_", "_", "R", "q_loop_grgat"],
24  ["q4", "x", "x", "R", "q4"],
   ["q5", "0", "0", "L", "q5"],
26  ["q5", "_", "_", "R", "q2"],
   ["q5", "x", "x", "L", "q5"],
28  ["q_accept", "0", "0", "R", "q_accept"],
   ["q_accept", "_", "_", "R", "q_accept"],
30  ["q_accept", "x", "x", "R", "q_accept"],
   ["q_loop_grgat", "0", "0", "R", "q_loop_grgat"],
32  ["q_loop_grgat", "_", "_", "R", "q_loop_grgat"],
   ["q_loop_grgat", "x", "x", "R", "q_loop_grgat"],
34  ["q_reject", "0", "0", "R", "q_loop_grgat"],
   ["q_reject", "_", "_", "R", "q_loop_grgat"],
36  ["q_reject", "x", "x", "R", "q_loop_grgat"]
   ],
38  "word":"0000"
}

```

As you can see this JSON output correctly represents the mapped string  $\langle M', w \rangle$  with the outputted  $M'$  having the same specifications and state diagram as drawn in Figure 6. Therefore the program has correctly mapped this example  $x = \langle M, w \rangle$  because  $x = \langle M, w \rangle \in A_{TM} \iff F(x) = \langle M', w \rangle \in HALT_{TM}$  and since  $\langle M, w \rangle \in A_{TM}$ , the program has correctly outputted  $F(x) = \langle M', w \rangle \in HALT_{TM}$ .

Note that `loop_state` /  $q_{loop}$  is `q_loop_grgat` because a random length-5 string is appended to `q_loop_` just in case `q_loop` is already a state in  $M$ .

### Negative Instance

For a negative instance of a string  $x = \langle M, w \rangle \notin A_{TM}$  where  $M$  is a Turing machine and  $w$  is a string over the Turing machine's input alphabet, take figure 3.8 from the textbook (Figure 5) with the input string  $w = \epsilon$ .

$\langle M, w \rangle \notin A_{TM}$  because the computation of  $M$  on  $w$  halts in  $q_{reject}$  through the following steps:

Tape: \_\_\_\_\_, State:  $q_1$ , Read pointer in position 0

Tape: \_\_\_\_\_, State:  $q_{reject}$ , Read pointer in position 1

Given this  $\langle M, w \rangle$ ,  $x$  is the following JSON string:

```
{
2  "states": ["q1", "q2", "q3", "q4", "q5", "q_accept", "q_reject"],
   "input_alphabet": ["0"],
4   "tape_alphabet": ["0", "x", "_"],
   "transitions": [
6     ["q1", "_", "_", "R", "q_reject"],
     ["q1", "x", "x", "R", "q_reject"],
8     ["q1", "0", "_", "R", "q2"],

10    ["q2", "_", "_", "R", "q_accept"],
     ["q2", "x", "x", "R", "q2"],
12    ["q2", "0", "x", "R", "q3"],

14    ["q3", "_", "_", "L", "q5"],
     ["q3", "x", "x", "R", "q3"],
16    ["q3", "0", "0", "R", "q4"],

18    ["q4", "_", "_", "R", "q_reject"],
     ["q4", "x", "x", "R", "q4"],
20    ["q4", "0", "x", "R", "q3"],

22    ["q5", "_", "_", "R", "q2"],
     ["q5", "x", "x", "L", "q5"],
24    ["q5", "0", "0", "L", "q5"],

26    ["q_accept", "_", "_", "R", "q_accept"],
     ["q_accept", "x", "x", "R", "q_accept"],
28    ["q_accept", "0", "0", "R", "q_accept"],

30    ["q_reject", "_", "_", "R", "q_reject"],
     ["q_reject", "x", "x", "R", "q_reject"],
32    ["q_reject", "0", "0", "R", "q_reject"]
   ],
34  "start_state": "q1",
   "accept_state": "q_accept",
36  "reject_state": "q_reject",
   "word": ""
38 }
```

Same as the positive example,  $M'$  should have the state diagram shown in Figure 6 when mapped with  $F$ .

$w$  should still be unchanged so we would get  $\langle M', w \rangle$  which is not in  $HALT_{TM}$  because the computation of  $M'$  on  $w$  goes as follows:

Tape: \_\_\_\_\_, State:  $q_1$ , Read pointer in position 0

Tape: \_\_\_\_\_, State:  $q_{loop}$ , Read pointer in position 1

Tape: \_\_\_\_\_, State:  $q_{loop}$ , Read pointer in position 2

Tape: \_\_\_\_\_, State:  $q_{loop}$ , Read pointer in position 3

This cycle would then continue on  $q_{loop}$  forever and  $M'$  would never halt on  $w$ , therefore making  $\langle M', w \rangle \notin HALT_{TM}$ .

Upon running the code on this JSON input, we get the following JSON output:

```
[michael@nixos:~/School/15th_Grade/CSE105/project]$ elixir lib/map_reduce.exs "${cat map_reduce_specs/power_of_two_zero_reject.json}"
{"accept_state":"q_accept","input_alphabet":["0"],"reject_state":"q_reject","start_state":"q1","states":["q1","q2","q3","q4","q5","q_accept","q_loop_xgujl","q_reject"],"tape_alphabet":["0","_","x"],"transitions":[["q1","0","_","R","q2"],["q1","_","_","R","q_loop_xgujl"],["q1","x","x","R","q_loop_xgujl"],["q2","0","_","R","q3"],["q2","_","_","R","q_loop_xgujl"],["q2","x","x","R","q3"],["q3","0","_","R","q4"],["q3","_","_","R","q_loop_xgujl"],["q3","x","x","R","q3"],["q4","0","_","R","q3"],["q4","_","_","R","q_loop_xgujl"],["q4","x","x","R","q4"],["q5","0","_","L","q5"],["q5","_","_","R","q2"],["q5","x","x","L","q5"],["q_accept","0","0","R","q_accept"],["q_accept","_","_","R","q_accept"],["q_accept","x","x","R","q_accept"],["q_loop_xgujl","0","0","R","q_loop_xgujl"],["q_loop_xgujl","_","_","R","q_loop_xgujl"],["q_loop_xgujl","x","x","R","q_loop_xgujl"],["q_reject","0","0","R","q_loop_xgujl"],["q_reject","_","_","R","q_loop_xgujl"],["q_reject","x","x","R","q_loop_xgujl"],"word":""}]
```

Figure 8: Running the code on the negative example

Note again how we use `cat` to avoid having to type the entire JSON input string and are able to read it from a file.

Here is the formatted JSON output for easier reading:

```
{
  "accept_state": "q_accept",
  "input_alphabet": [
    "0"
  ],
  "reject_state": "q_reject",
  "start_state": "q1",
  "states": ["q1", "q2", "q3", "q4", "q5", "q_accept", "q_loop_xgujl",
    "q_reject"],
  "tape_alphabet": ["0", "_", "x"],
  "transitions": [
    ["q1", "0", "_", "R", "q2"],
    ["q1", "_", "_", "R", "q_loop_xgujl"],
    ["q1", "x", "x", "R", "q_loop_xgujl"],
    ["q2", "0", "x", "R", "q3"],
    ["q2", "_", "_", "R", "q_accept"],
    ["q2", "x", "x", "R", "q2"],
    ["q3", "0", "0", "R", "q4"],
    ["q3", "_", "_", "L", "q5"],
    ["q3", "x", "x", "R", "q3"],
    ["q4", "0", "x", "R", "q3"],
    ["q4", "_", "_", "R", "q_loop_xgujl"],
    ["q4", "x", "x", "R", "q4"],
    ["q5", "0", "0", "L", "q5"],
    ["q5", "_", "_", "R", "q2"],
    ["q5", "x", "x", "L", "q5"],
    ["q_accept", "0", "0", "R", "q_accept"],
    ["q_accept", "_", "_", "R", "q_accept"],
    ["q_accept", "x", "x", "R", "q_accept"],
    ["q_loop_xgujl", "0", "0", "R", "q_loop_xgujl"],
    ["q_loop_xgujl", "_", "_", "R", "q_loop_xgujl"],
    ["q_loop_xgujl", "x", "x", "R", "q_loop_xgujl"],
    ["q_reject", "0", "0", "R", "q_loop_xgujl"],
    ["q_reject", "_", "_", "R", "q_loop_xgujl"],
    ["q_reject", "x", "x", "R", "q_loop_xgujl"]
  ],
  "word": ""
}
```

As you can see this JSON output correctly represents the mapped string  $\langle M', w \rangle$  with the outputted  $M'$  having the same specifications and state diagram as drawn in Figure 6. Therefore the program has correctly

mapped this example  $x = \langle M, w \rangle$  because  $x = \langle M, w \rangle \in A_{TM} \iff F(x) = \langle M', w \rangle \in HALT_{TM}$  means that since  $x \notin A_{TM}$ , the program has correctly outputted  $F(x) = \langle M', w \rangle \notin HALT_{TM}$ .

Note that `loop_state / qloop` is `qloop_xgujl` because a random length-5 string is appended to `qloop` just in case `qloop` is already a state in  $M$ .

## Video Explanation

Watch the video [here](#)

## Code

If you'd like to read the code in a better viewing format than PDF, visit the file on [GitHub](#)

```
defmodule Project.MapReduce do
2   @doc """
    Parses a json string representing <M, w> where M is a Turing machine and
4   w is a string over M's input alphabet. Parses the JSON string to
        validate M and
        to get the formal definition of states,
6   input + tape alphabet, transition function, start state, and accept
        state, and
        reject state while also validating the input string w is valid.
8
    ## JSON Turing Machine Format
10  {
        states: string[]
12  input_alphabet: char[]
        tape_alphabet: char[]
14  transitions: tuple<string, string, string, string, string>[]
        start_state: string
16  accept_state: string
        reject_state: string
18  word: string
    }
20  states is an array of states in the Turing machine
    input_alphabet is an array of characters in the Turing machines input
        alphabet
22  tape_alphabet is an array of characters in the Turing machines tape
        alphabet
    transitions is an array of 5-tuples defining
24  the Turing machine's transition function where the first string is the
        input state, the second string is the character being read,
26  and the third string is the character to write, the fourth string
        is the direction to move the tape head ('R' | 'L') and the fifth string
            is the
28  output state.
        start_state is the start state of the Turing machine
30  accept_state is the accepting state of the Turing machine
        reject_state is the rejecting state of the Turing machine
32  word is the string input to the Turing machine

34  Returns the Turing machine with the arrays cast to sets and the
        transitions as an
        adjacency list as well as the input string w
```

```

36  """
37  def parse_tm(input) do
38    %{
39      "states" => states,
40      "input_alphabet" => input_alphabet,
41      "tape_alphabet" => tape_alphabet,
42      "transitions" => transitions,
43      "start_state" => start_state,
44      "accept_state" => accept_state,
45      "reject_state" => reject_state,
46      "word" => word
47    } = :json.decode(input)
48
49    states = MapSet.new(states)
50    input_alphabet = MapSet.new(input_alphabet)
51    tape_alphabet = MapSet.new(tape_alphabet)
52    head_directions = MapSet.new(["L", "R"])
53
54    # map the transition [start state, read character, write character,
55      tape direction, end state]
56    transitions = Enum.reduce(transitions, %{}, fn [start, read, write,
57      dir, dest], acc ->
58      Map.update(
59        acc,
60        start,
61        %{ read => {write, dir, dest} },
62        fn existing ->
63          # if this state already has an existing transition with the
64            read character
65          # raise an error
66          if Map.has_key?(existing, read) do
67            raise "A state cannot have two transitions with the same read
68              character"
69          end
70          Map.put(existing, read, {write, dir, dest})
71        end
72      )
73    end)
74
75    # assert that the input alphabet is a subset of the tape alphabet
76    if not MapSet.subset?(input_alphabet, tape_alphabet) do
77      raise "The input alphabet must be a subset of the tape alphabet"
78    end
79
80    # assert that the accept state is in the states
81    if not MapSet.member?(states, accept_state) do
82      raise "The accept state must be a member of the states array"
83    end
84
85    # assert that the reject state is in the states
86    if not MapSet.member?(states, reject_state) do
87      raise "The reject state must be a member of the states array"
88    end
89
90    # assert that the start state is in the states
91    if not MapSet.member?(states, start_state) do

```



```

86     raise "The start state must be a member of the states array"
87   end
88   # assert that every state has transition rules
89   if not MapSet.equal?(states, transitions |> Map.keys |> MapSet.new) do
90     raise "Every state must have transitions defined"
91   end
92   # assert that every state has a transition for every member of the
93   # tape alphabet
94   transitions
95     |> Map.values
96     |> Enum.each(fn rules ->
97       # assert that the state's rules has a rule for each tape character
98       if not MapSet.equal?(tape_alphabet, rules |> Map.keys |>
99         MapSet.new) do
100         raise "Each state must have a transition defined for each
101           member of the tape alphabet"
102       end
103       # assert that every destination in the rules is a valid state
104       if not MapSet.subset?(rules |> Map.values |> Enum.map(fn { _, _,
105         dest } -> dest end) |> MapSet.new, states) do
106         raise "The destination of a state transition must be a valid
107           state in the state array"
108       end
109       # assert that every write character is valid in tape alphabet
110       if not MapSet.subset?(rules |> Map.values |> Enum.map(fn { write,
111         _, _ } -> write end) |> MapSet.new, tape_alphabet) do
112         raise "The character being written to tape must be a member of
113           the tape alphabet"
114       end
115       # assert that every tape direction is either L or R
116       if not MapSet.subset?(rules |> Map.values |> Enum.map(fn { _,
117         dir, _ } -> dir end) |> MapSet.new, head_directions) do
118         raise "The tape head direction must be either 'R' or 'L'"
119       end
120     end)
121
122   # now that tm is valid, assert that the input string only contains
123   # char from input alphabet
124   word = String.graphemes(word)
125   if not MapSet.subset?(word |> MapSet.new, input_alphabet) do
126     raise "The word must only contain characters form the input
127       alphabet"
128   end
129
130   {states, input_alphabet, tape_alphabet, transitions, start_state,
131     accept_state, reject_state, word}
132 end
133
134 @doc """
135 The inverse of parse_tm where it takes the outputs and can serialize
136 them back to the JSON format specified above.
137 """
138 def serialize_tm({states, input_alphabet, tape_alphabet, transitions,
139   start_state, accept_state, reject_state, word}) do

```

```

128   %{
129     "states" => states |> MapSet.to_list,
130     "input_alphabet" => input_alphabet |> MapSet.to_list,
131     "tape_alphabet" => tape_alphabet |> MapSet.to_list,
132     "transitions" => transitions
133       |> Enum.flat_map(fn {start, rules} ->
134         Enum.map(rules, fn { read, { write, dir, dest } } ->
135           [start, read, write, dir, dest]
136         end)
137       end),
138     "start_state" => start_state,
139     "accept_state" => accept_state,
140     "reject_state" => reject_state,
141     "word" => word |> Enum.join
142   } |> :json.encode
143 end
144
145 @doc """
146 For usage in generating the looping state in the mapping reduction.
147 Given the set of existing states and a prefix, returns a new state
148 "<prefix>_<random 5 characters>" that doesn't already exist in the
149 set of states.
150 """
151 def new_state(states, prefix) do
152   random_state = for _ <- 1..5, into: "#{prefix}_", do:
153     <<Enum.random(?a..?z)>>
154   if MapSet.member?(states, random_state) do
155     new_state(states, prefix)
156   else
157     random_state
158   end
159 end
160
161 @doc """
162 Maps the Turing machine acceptance problem to the halting problem
163 Given the input <M,w>, maps to <M', w> with the technique discussed in
164 class where if a computation were ever to go to the reject state,
165 it would loop instead. Outputs <M', w> in the same input string format
166 if the input is valid.
167 """
168 def map_reduce(input) do
169   {
170     states,
171     input_alphabet,
172     tape_alphabet,
173     transitions,
174     start_state,
175     accept_state,
176     reject_state,
177     word,
178   } = parse_tm(input)
179   # make a new loop state
180   loop_state = new_state(states, "q_loop")
181   states = MapSet.put(states, loop_state)

```

```

182   # for every edge that goes to the reject state, go to a new loop state
transitions = transitions
184   |> Enum.reduce(%{}, fn {start, rules}, acc ->
      rules = Enum.reduce(rules, %{}, fn { read, { write, dir, dest }
        }, acc ->
186         if dest == reject_state do
          # if going to reject state, redirect to loop state
188           Map.put(acc, read, {write, dir, loop_state})
        else
          # if not going to reject state, leave it unchanged
          Map.put(acc, read, {write, dir, dest })
192         end
      end)
194   Map.put(acc, start, rules)
end)
196   |> Map.put(loop_state, Enum.reduce(tape_alphabet, %{}, fn char, acc
    ->
      # now add in the transitions for the loop state, basically for
198      # every character in the tape alphabet loop back to loop_state
      # leave the tape unchanged and move the tape head right
200      Map.put(acc, char, {char, "R", loop_state}))
end))

202
204   # output the formatted new turing machine
serialize_tm({states, input_alphabet, tape_alphabet, transitions,
  start_state, accept_state, reject_state, word})
end

206
@doc"""
208 Calls map_reduce with the first command line argument to this script.
If an error is thrown that means parsing failed, aka <M, w> is not a
  valid
210 representation of a Turing machine and input string over M's input
alphabet. In this case, const_tm is outputted which is not a member
212 of HALT_TM. If no error is thrown it returns the mapping of <M, w> from
A_TM to HALT_TM.
214 """
def call do
216   try do
     map_reduce(System.argv() |> Enum.at(0, ""))
218   rescue
     _ -> {
220     # if an error is thrown during the map reduction that means
      parsing failed
      # so lets return a constant which is not in halt_tm. we will use
      a TM
222     # that loops on the start state forever
      MapSet.new(["q_start", "q_acc", "q_rej"]),
224     MapSet.new(["0"]),
      MapSet.new(["0", "_"]),
226     %{
      "q_start" => %{
228       "0" => { "0", "R", "q_start" },

```

```

230         "_" => { "_", "R", "q_start" }
231     },
232     "q_acc" => %{
233         "0" => { "0", "R", "q_acc" },
234         "_" => { "_", "R", "q_acc" }
235     },
236     "q_rej" => %{
237         "0" => { "0", "R", "q_rej" },
238         "_" => { "_", "R", "q_rej" }
239     }
240 },
241 "q_start",
242 "q_acc",
243 "q_rej",
244 ""
245 } |> serialize_tm()
246 end
247 end
248 end
Project.MapReduce.call |> IO.puts

```