Proteus is Turing Complete

Isaiah

Requirements for TC

- To be TC you require:
  - Basic Arithmetic/Binary Operators (+,-,*,/)
  - Conditional Statements (if)
  - Looping (while)
  - Readable and Writable Memory (store)
- We lack the ability to overwrite a value of a complex structure like an array
- Instead we can utilize the value of a variable

Variable = ???

- However, the variable is what I’m most unsure of. Firstly,
  - We are able to read its value dynamically
  - We are able to write its value dynamically
- The issue is how to understand what the variable means to us
  - Does it store the infinite tape of 0’s and 1’s in a classical TM?
  - Does it represent the output of some problem we give the machine to solve?
  - How do we utilize it to show TC explicitly?

Variable = 42 => Profit $$$

- The variable is just a representative of what we will be storing based on the computation of the given program/machine
  - I.e. it is **variable** and relies on the objective/goal
- Thus, we have the building blocks of showing TC.
- The problem is how…

Different Approaches

Math/Engineering Approach

Approach 1

- We utilize Math and Engineering notation for State Machines to show the classical understanding of TC for this language.
- This is explicit and highly technical, but concise.
- This approach also utilizes the usage of State Machines and the notation for them.
- Src: “Turing Completeness and Sid Meier’s Civilization”

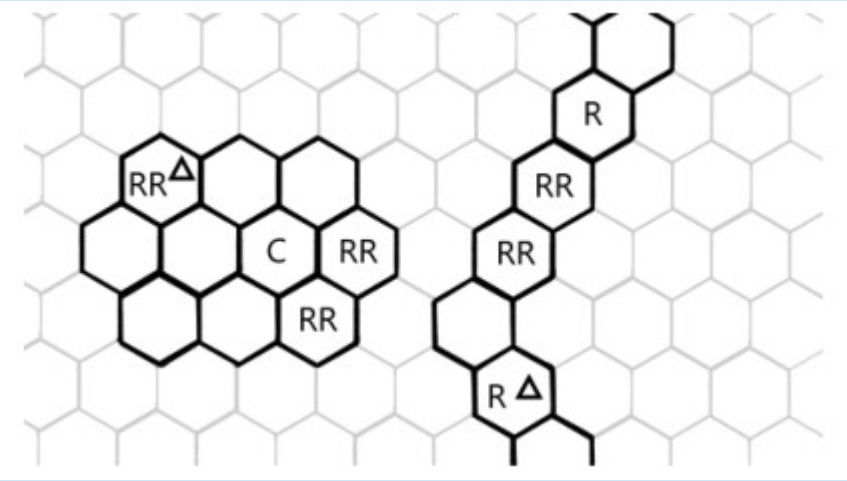
$\langle Q, \Gamma, \Sigma, \delta, b, q_0, F \rangle$, where:

1) Q is a set of possible *states* that can be taken internally by the Turing machine. Namely, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of final (or accepting) states.
2) Γ is the set of tape *symbols* on which the Turing machine can perform read/write operations. In particular, $b \in \Gamma$ is the blank symbol, and $\Sigma \subseteq \Gamma$, $b \notin \Sigma$ is the set of symbols which may be initially placed in the tape.
3) $\delta : (Q \setminus F) \times \Gamma \nrightarrow Q \times \Gamma \times \{L, R\}$ is the *transition function*. It is a partial function that takes in all states from the machine, along with the current read from the tape; it determines the next state to be taken by the Turing machine, as well as whether to move the head left (L) or right (R) along the tape. Not moving the tape can be trivially encoded in this definition.

Approach 1

Civ:BE state (Δ; *tape read*)	Command to Workers (*tape*; *state*)	BB-3 TM
0; No Improvement	Build a Road and move R; Build a Terrascape	$q_0 0; 1Rq_1$
0; Road	Build a Road and move L; Build 2 Terrascapes	$q_0 1; 1Lq_2$
1; No Improvement	Build a Road and move L; Remove a Terrascape	$q_1 0; 1Lq_0$
1; Road	Build a Road and move R; No build	$q_1 1; 1Rq_1$
2; No Improvement	Build a Road and move L; Remove a Terrascape	$q_2 0; 1Lq_1$
2; Road	HALT	$q_2 1; 1R\ HALT$

State/Tape Reading



Map Layout of Tape

Then the following is a $(10, 3)$-UTM:

$$Q = \{0, \ldots, 9\} \quad (3)$$

$$\Gamma = \{\text{No Improvement}, \text{Road}, \text{Railroad}\} \quad (4)$$

where $q_i \in Q$ is the total number of Railroads in the nine hexes, and the transition function δ is as described in Table III.

Description of States, Transition functions, Symbols

Wordy/Game-ified Approach

Approach 2

- We utilize words and describe what each of the transitions/reading/writing looks like in a TM.
- This would result in a much longer paper/thesis, but more easily readable to the layman.
- This approach is more “game-ified” and thus easier to digest. Think of a 1 or 2 player game.
- Src: “Magic: The Gathering is Turing Complete”

Rogozhin's program				Card text
q_1	1	c_2	Lq_1	Whenever an Aetherborn dies, create a 2/2 white Sliver
q_1	$\overrightarrow{1}$	$\overleftarrow{1}_1$	Rq_1	Whenever a Basilisk dies, create a 2/2 green Elf
q_1	$\overleftarrow{1}$	c_2	Lq_1	Whenever a Cephalid dies, create a 2/2 white Sliver
q_1	$\overrightarrow{1}_1$	1	Rq_1	Whenever a Demon dies, create a 2/2 green Aetherborn
q_1	$\overleftarrow{1}_1$	$\overrightarrow{1}_1$	Lq_1	Whenever an Elf dies, create a 2/2 white Demon
q_1	b	$\overleftarrow{b}$	Rq_1	Whenever a Faerie dies, create a 2/2 green Harpy
q_1	$\overrightarrow{b}$	$\overleftarrow{b}_1$	Rq_1	Whenever a Giant dies, create a 2/2 green Juggernaut
q_1	$\overleftarrow{b}$	b	Lq_1	Whenever a Harpy dies, create a 2/2 white Faerie
q_1	$\overrightarrow{b}_1$	b	Rq_1	Whenever an Illusion dies, create a 2/2 green Faerie
q_1	$\overleftarrow{b}_1$	$\overrightarrow{b}_1$	Lq_1	Whenever a Juggernaut dies, create a 2/2 white Illusion
q_1	b_2	b_3	Lq_2	Whenever a Kavu dies, create a tapped 2/2 white Leviathan
q_1	b_3	$\overrightarrow{b}_1$	Lq_2	Whenever a Leviathan dies, create a tapped 2/2 white Illusion
q_1	c	$\overrightarrow{1}$	Lq_2	Whenever a Myr dies, create a tapped 2/2 white Basilisk
q_1	$\overrightarrow{c}$	$\overleftarrow{c}$	Rq_1	Whenever a Noggle dies, create a 2/2 green Orc
q_1	$\overleftarrow{c}$	$\overrightarrow{c}_1$	Lq_1	Whenever an Orc dies, create a 2/2 white Pegasus
q_1	$\overrightarrow{c}_1$	$\overleftarrow{c}_1$	Rq_2	Whenever a Pegasus dies, create a tapped 2/2 green Rhino
q_1	$\overleftarrow{c}_1$	$HALT$		Whenever a Rhino dies, create a 2/2 blue Assassin
q_1	c_2	$\overleftarrow{1}$	Rq_1	Whenever a Sliver dies, create a 2/2 green Cephalid
q_2	1	$\overleftarrow{1}$	Rq_2	Whenever an Aetherborn dies, create a 2/2 green Cephalid
q_2	$\overrightarrow{1}$	$\overleftarrow{1}$	Rq_2	Whenever a Basilisk dies, create a 2/2 green Cephalid
q_2	$\overleftarrow{1}$	$\overrightarrow{1}$	Lq_2	Whenever a Cephalid dies, create a 2/2 white Basilisk
q_2	$\overrightarrow{1}_1$	$\overleftarrow{1}_1$	Rq_2	Whenever a Demon dies, create a 2/2 green Elf
q_2	$\overleftarrow{1}_1$	1	Lq_2	Whenever an Elf dies, create a 2/2 white Aetherborn
q_2	b	b_2	Rq_1	Whenever a Faerie dies, create a tapped 2/2 green Kavu
q_2	$\overrightarrow{b}$	$\overleftarrow{b}$	Rq_2	Whenever a Giant dies, create a 2/2 green Harpy
q_2	$\overleftarrow{b}$	$\overrightarrow{b}$	Lq_2	Whenever a Harpy dies, create a 2/2 white Giant
q_2	$\overrightarrow{b}_1$	$\overleftarrow{b}_1$	Rq_2	Whenever an Illusion dies, create a 2/2 green Juggernaut
q_2	$\overleftarrow{b}_1$	$\overrightarrow{b}$	Lq_2	Whenever a Juggernaut dies, create a 2/2 white Giant
q_2	b_2	b	Rq_1	Whenever a Kavu dies, create a tapped 2/2 green Faerie
q_2	b_3	$\overleftarrow{b}_1$	Rq_2	Whenever a Leviathan dies, create a 2/2 green Juggernaut
q_2	c	$\overleftarrow{c}$	Rq_2	Whenever a Myr dies, create a 2/2 green Orc
q_2	$\overrightarrow{c}$	$\overleftarrow{c}$	Rq_2	Whenever a Noggle dies, create a 2/2 green Orc
q_2	$\overleftarrow{c}$	$\overrightarrow{c}$	Lq_2	Whenever an Orc dies, create a 2/2 white Noggle
q_2	$\overrightarrow{c}_1$	c_2	Rq_2	Whenever a Pegasus dies, create a 2/2 green Sliver
q_2	$\overleftarrow{c}_1$	c_2	Lq_1	Whenever a Rhino dies, create a tapped 2/2 white Sliver
q_2	c_2	c	Lq_2	Whenever a Sliver dies, create a 2/2 white Myr

Approach 2

B. The Controller

Control instructions in a Turing machine are represented by a table of conditional statements of the form "if the machine is in state s, and the last read cell is symbol k, then do such-and-such." Many *Magic* cards have triggered abilities which can function like conditional statements. The two we shall use are **Rotlung Reanimator** ("Whenever Rotlung Reanimator or another Cleric dies, create a 2/2 black Zombie creature token") and **Xathrid Necromancer** ("Whenever Xathrid Necromancer or another Human creature you control dies, create a tapped 2/2 black Zombie creature token"). We will use both, and the difference between tapped and untapped creature tokens will contribute to the design of the Turing machine.

Explicit description of what each component of TC requires and how to achieve it

B. Reading the Current Cell

On the first turn of the cycle, Alice is forced to cast **Infest**, "All creatures get –2/–2 until end of turn." This kills one creature: the tape token at the position of the current read head, controlled by Bob. This will cause precisely one creature of Bob's to trigger – either a **Rotlung Reanimator** or a **Xathrid Necromancer**. Which precise one triggers is based

How each component is represented in the game (with each card)

We similarly have seventeen more **Rotlung Reanimator**s encoding the rest of the q_1 program from [14]. Between them they say:

1) Whenever an *Aetherborn* dies, create a 2/2 *white Sliver*.
2) Whenever a *Basilisk* dies, create a 2/2 *green Elf*.

Whenever a ... dies, create a 2/2 ...

18) Whenever a *Sliver* dies, create a 2/2 *green Cephalid*.

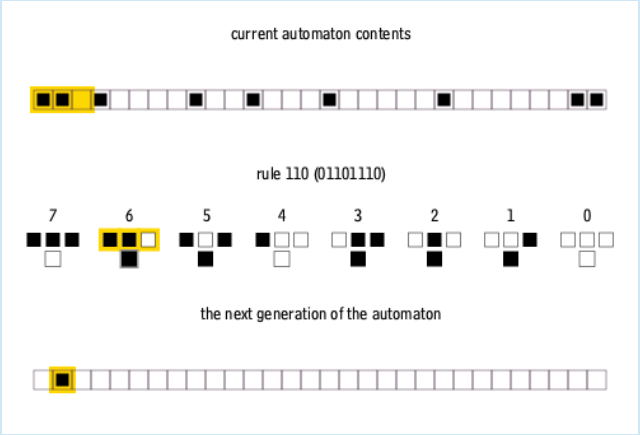
See Table II for the full encoding of the program.

Show the different outcomes and states

Coding

Approach 3

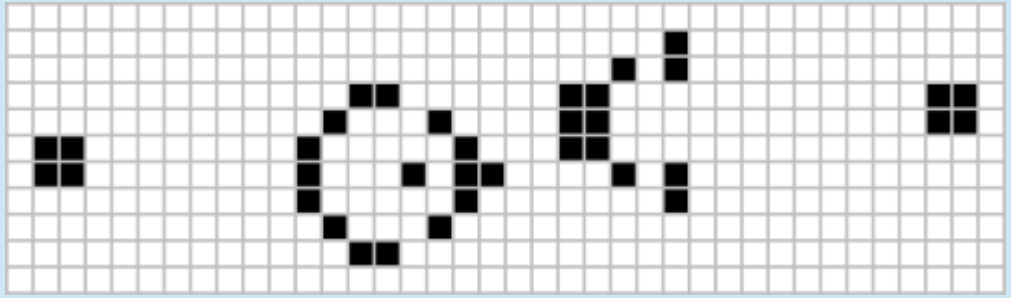
- We will rely on utilizing the language itself (abstracting from the grammar) by creating program(s) to demonstrate TC
- These programs would demonstrate TC problems like Rule 110 or Conway's Game of Life
- This is probably the most difficult and time consuming, but simplest to explain
  - "B/C this is TC and Proteus can do it, then Proteus is TC"



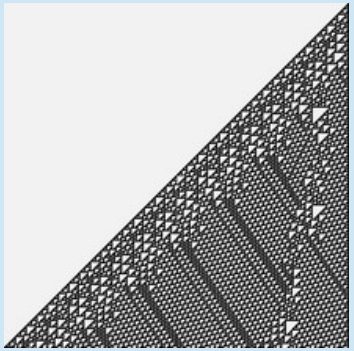
Approach 3

Command	Description
`>`	Move the pointer to the right
`<`	Move the pointer to the left
`+`	Increment the memory cell at the pointer
`-`	Decrement the memory cell at the pointer
`.`	Output the character signified by the cell at the pointer
`,`	Input a character and store it in the cell at the pointer
`[`	Jump past the matching `]` if the cell at the pointer is 0
`]`	Jump back to the matching `[` if the cell at the pointer is nonzero

Brainfuck Interpreter



Conway's Game of Life

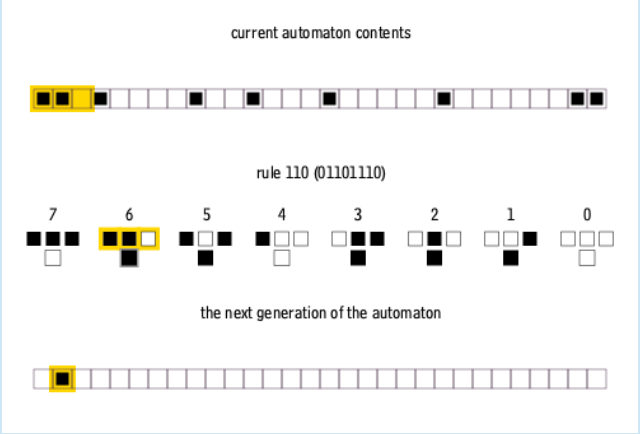


Rule 110

What next?

Proof

- I think the simplest would be to do Approach 3. It provides the most straightforward understanding to showing how Proteus is TC.
- I however would prefer to do a mixture of the 3:
  - I would like to show Rule 110 using a combination of State Machines and functions (Approach 3)
  - Describe how it works using the more mathematical and engineering notation for the state machines used (Approach 1)
  - Talk through the process of how it works step by step; i.e. simulating/running the program with words (Approach 2)



$Q =$ {Start, Stop}

$\Gamma =$ {0, 1} => {Off, On}

Proof

- Specifically, I would like to avoid using the mathematical notation heavily. One of the things that stumped me for a long time was trying to understand how to apply the math notation (delta, tau, lambda, etc.) to our grammar.
- It would be easier to describe “physical” state machines (each state machine could have an LED which represents on/off) which would be coded using Proteus.
  - This can be used for Conway’s Game of Life or Rule 110 (I think the latter is cooler, but CGoL would probably be easier in design I think)
  - I would like to retain the notation for the state machines however, as it makes it more “science-y”

Proof

- The last part (Approach 2) is really just optional and considered padding/extra stuff at the end to fill pages or something. It would basically be just for further explanation/discussion I think.
- I’m not entirely sure how to structure the Thesis and whatnot so I’m eager to hear your feedback.

Mock Diagram

My Questions

- Does this sound reasonable to aim for in a semester?
  - Is the scope too large? Am I too ambitious? Is it not enough for a paper/thesis?
- How many citations needed? What number of pages to aim for in the thesis/paper?
- How do I run Proteus code on Windows or MacOS?
  - Does the compiler v1 or v2 work as of right now? Are there any compilation things I should know of in particular?
- Any final thoughts?