12th LogicComputation_Week 2_Lesson 3: A Universal Turing Machine Name _____

key 1 Per. _____ Date _____

Lesson Overview:

Today you are mainly reading lecture notes. There is a question or two to answer at the end, but the focus of today is to get some understanding and appreciation for a radical new idea. The idea we are going to learn about today comes in two parts:

- It is possible to take any Turing Machine diagram and represent it with a string of symbols. In other words, rather than having arrows drawn between different bubbles, you could encode the states and transitions with some kind of symbolic description like: "0L,1R,2Y,3N;0-1#:#,1-2#:#,1-1a:b,1-3b:b".
- This is where things get meta. That symbolic description of a Turing Machine could be written onto the tape of another Turing Machine! There could be a special-purpose TM that is meant to read and decide something about another TM that is encoded onto its tape. Or maybe there could be an all-purpose TM that could simulate any TM written onto its tape.

I hope that hurt your head a little! Do you see what a wild idea that is? We will set up and trace through the implications of this idea in this lesson. Ultimately we will see that this is what makes computers and computer programs possible.

Main Reading: The Universal Turing Machine

Much of this material is taken from https://introcs.cs.princeton.edu/java/53universality/.

By this point in the year, we have seen several remarkable things:

1. It is possible to build a physical model, using circuits or other mechanical devices, for boolean logic, which allows us to implement some subset of algorithms, e.g. “Output the result of a Rock, Paper, Scissors game given the input of the players.

2. By adding memory storage to the concept, we get a Turing Machine, which allows us to implement lots more algorithms.

We should note that, at least in the example of circuits, the “what” that our little computers can do is actually quite broad. It all depends on how you interpret the “on” and “off” of the input switches. For example:

1. “On, Off, Off, Off, On, Off” could represents Player 1 playing “Rock” and Player 2 playing “Paper” in the RPS game. (Here the first three inputs are for Player 1 and the second three for Player 2.)

2. The series of On’s and Off’s could represent binary numbers, which would allow us to computer arithmetic.

3. The series of On’s and Off’s could represent a unique binary code that corresponds to a typed character, which the allows for the processing of text (words, etc.).

4. The series of On’s and Off’s could represent actual lights (let’s call them “pixels”), which would allow for the display of graphics.

All of this can be handled by a Turing Machine. In fact as we will see, the concept of a Turing Machine is so powerful that it can accomplish *any* algorithm. Informally, an algorithm is a step-by-step process for solving a problem. But Computer Scientists *define* an algorithm as “the processing of information that can be done by a Turing Machine.”

There is, however, one major drawback to all of this. While Turing Machines are very powerful as a group, any given Turing Machine can only perform *one* algorithm. It exists only to solve *one* problem. Another way of saying this is that a Turing Machine is akin to a computer program, not to a computer (at least as we understand the term).

Think of the RPS “computer” you built. It *only* played RPS. That’s it. If you want a device that adds numbers together, you need to build a separate set of circuitry. In fact, you need a third one to multiply numbers, because multiplying numbers is a different algorithm (i.e. Turing Machine) that adding numbers.

In fact, for every task you can think of, you would need a brand new computer. If this were the end of the story,

computers would have never taken off. Only a few people would own a few kinds of devices. (Maybe an accountant would own an addition device and a [separate] subtraction device, maybe a few kids would own a RPS computer, etc.)

Alan Turing had a bold vision.

The Vision: What if we could conceive of a Turing Machine and its tape as "input." Then we could try to create a separate Turing Machine that takes this input and produces the same output as the original Turing Machine. ***And it would do this no matter what Turing Machine and tape it receives.***

Let's consider three separate Turing Machines and input tapes: TMA-x (Turing Machine A with tape x), TMA-y (Turing Machine A with tape y), and TMB-z (Turing Machine B with tape z). Further, let's suppose that the result of TMA-x (or Turing Machine A starting with tape x) is the tape X. Similarly the result of TMA-y is Y, and the result of TMB-z is Z.

Turing wanted to (1) code these Machine-Tape combinations *onto a new Tape itself*, and (2) construct a new nifty machine called "Universal Turing Machine", or UTM for short. His vision for the UTM was bold:

1. When it received the tape that encoded TMA-x, it would output X.
2. When it received the tape that encoded TMA-y, it would output Y.
3. When it received the tape that encoded TMB-z, it would output Z.

Actually, it is more bold than that. When it receives *any* Turing Machine with accompanying tape, (TM-t), that produces the output T, *it too would produce T*.

If this were possible, you would only need a single machine: the UTM. All other "tasks" would not need to be separate machines, but could actually be "coded machine", or "code" … or computer programs.

But how to do the critical steps?

1. How do we encode a Turing Machine and its input tape as data on some other input tape?
2. How do we build the Universal Turing Machine that does what is required?

Step 1: Encoding a Turing Machine

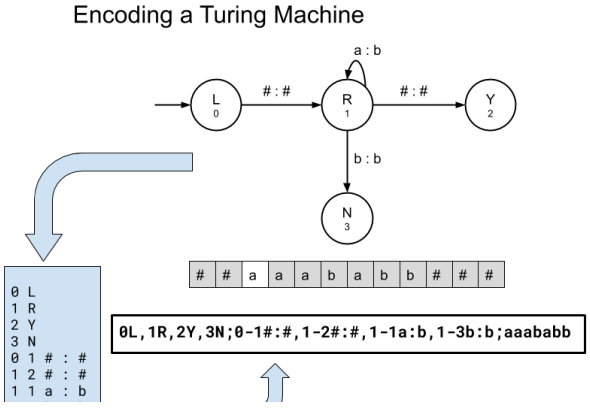
To encode a Turing Machine completely, we must capture several pieces of information.

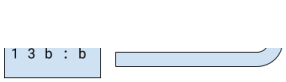
- First we enumerate all of the states.In addition to labeling the states with "R", "L", "Y", etc. we also give each state a unique number. We use 0 to represent the initial state.
- Next we list every state transition. We describe where the arrow begins and ends with state numbers. In addition to that we have to describe the read : write instruction on the arrow.
- Finally, we can also include the Turing Machine's own tape by simply appending it to the end.

This results in a single string of symbols that describes a Turing Machine and its input.

Study the example on the next slide to see how this is done.

Step 1: Encoding a Turing Machine





The result: "
0L,1R,2Y,3N;0-1#:#,1-2#:#,1-1a:b,1-3b:b;aaababb" tells us everything we needed to know about that Turing Machine. If we needed to, we could redraw the diagram just from this encoding.

Though, as mentioned earlier, Alan Turing had a unique motivation for encoding a TM as a data string. He wanted to place this data on the input tape of another Turing Machine.

He saw that it was possible to design a Universal Turing Machine that could take in one of these encodings and exactly simulate what it would do as it processed its data.

Step 2: Build a Universal Turing Machine

The UTM sounds too good to be true. It can simulate the behavior of any individual TM. In modern terms, the UTM is an *interpreter* - it performs a step-by-step simulation of any Turing machine. Program and data are the same thing - a program (Turing machine) is nothing more than a sequence of symbols that looks like any other data.

It is beyond the scope of what we can do here to describe a specific implementation of a UTM. Building a UTM appears to be a daunting task. Alan Turing described the first such Turing machine in his 1937 paper. In 1962, Marvin Minsky discovered a remarkable 7-state UTM that executes using a four-symbol alphabet, but it is rather complicated to describe. As recently as 1996, one was discovered with only 4 states and 6 symbols.

1. **Short Answer Response**

   Write one short paragraph (4-8 sentences) that answers the following questions: What makes the Universal Turing Machine different from any Turing Machine we have seen so far? How are the details of a Turing Machine able to be encoded on a memory tape? (1 pt)

12th LogicComputation_Week 2_Lesson 3: A Universal Turing Machine **ANSWER KEY 1**

key 1

Lesson Overview:

Today you are mainly reading lecture notes. There is a question or two to answer at the end, but the focus of today is to get some understanding and appreciation for a radical new idea. The idea we are going to learn about today comes in two parts:

- It is possible to take any Turing Machine diagram and represent it with a string of symbols. In other words, rather than having arrows drawn between different bubbles, you could encode the states and transitions with some kind of symbolic description like: "0L,1R,2Y,3N;0-1#:#,1-2#:#,1-1a:b,1-3b:b".
- This is where things get meta. That symbolic description of a Turing Machine could be written onto the tape of another Turing Machine! There could be a special-purpose TM that is meant to read and decide something about another TM that is encoded onto its tape. Or maybe there could be an all-purpose TM that could simulate any TM written onto its tape.

I hope that hurt your head a little! Do you see what a wild idea that is? We will set up and trace through the implications of this idea in this lesson. Ultimately we will see that this is what makes computers and computer programs possible.

Main Reading: The Universal Turing Machine

Much of this material is taken from https://introcs.cs.princeton.edu/java/53universality/.

By this point in the year, we have seen several remarkable things:

1. It is possible to build a physical model, using circuits or other mechanical devices, for boolean logic, which allows us to implement some subset of algorithms, e.g. “Output the result of a Rock, Paper, Scissors game given the input of the players.

2. By adding memory storage to the concept, we get a Turing Machine, which allows us to implement lots more algorithms.

We should note that, at least in the example of circuits, the “what” that our little computers can do is actually quite broad. It all depends on how you interpret the “on” and “off” of the input switches. For example:

1. “On, Off, Off, Off, On, Off” could represents Player 1 playing “Rock” and Player 2 playing “Paper” in the RPS game. (Here the first three inputs are for Player 1 and the second three for Player 2.)

2. The series of On’s and Off’s could represent binary numbers, which would allow us to computer arithmetic.

3. The series of On’s and Off’s could represent a unique binary code that corresponds to a typed character, which the allows for the processing of text (words, etc.).

4. The series of On’s and Off’s could represent actual lights (let’s call them “pixels”), which would allow for the display of graphics.

All of this can be handled by a Turing Machine. In fact as we will see, the concept of a Turing Machine is so powerful that it can accomplish *any* algorithm. Informally, an algorithm is a step-by-step process for solving a problem. But Computer Scientists *define* an algorithm as “the processing of information that can be done by a Turing Machine.”

There is, however, one major drawback to all of this. While Turing Machines are very powerful as a group, any given Turing Machine can only perform *one* algorithm. It exists only to solve *one* problem. Another way of saying this is that a Turing Machine is akin to a computer program, not to a computer (at least as we understand the term).

Think of the RPS “computer” you built. It *only* played RPS. That’s it. If you want a device that adds numbers together, you need to build a separate set of circuitry. In fact, you need a third one to multiply numbers, because multiplying numbers is a different algorithm (i.e. Turing Machine) that adding numbers.

In fact, for every task you can think of, you would need a brand new computer. If this were the end of the story,

computers would have never taken off. Only a few people would own a few kinds of devices. (Maybe an accountant would own an addition device and a [separate] subtraction device, maybe a few kids would own a RPS computer, etc.)

Alan Turing had a bold vision.

The Vision: What if we could conceive of a Turing Machine and its tape as "input." Then we could try to create a separate Turing Machine that takes this input and produces the same output as the original Turing Machine. ***And it would do this no matter what Turing Machine and tape it receives.***

Let's consider three separate Turing Machines and input tapes: TMA-x (Turing Machine A with tape x), TMA-y (Turing Machine A with tape y), and TMB-z (Turing Machine B with tape z). Further, let's suppose that the result of TMA-x (or Turing Machine A starting with tape x) is the tape X. Similarly the result of TMA-y is Y, and the result of TMB-z is Z.

Turing wanted to (1) code these Machine-Tape combinations *onto a new Tape itself*, and (2) construct a new nifty machine called "Universal Turing Machine", or UTM for short. His vision for the UTM was bold:

1. When it received the tape that encoded TMA-x, it would output X.
2. When it received the tape that encoded TMA-y, it would output Y.
3. When it received the tape that encoded TMB-z, it would output Z.

Actually, it is more bold than that. When it receives *any* Turing Machine with accompanying tape, (TM-t), that produces the output T, *it too would produce T*.

If this were possible, you would only need a single machine: the UTM. All other "tasks" would not need to be separate machines, but could actually be "coded machine", or "code" … or computer programs.

But how to do the critical steps?

1. How do we encode a Turing Machine and its input tape as data on some other input tape?
2. How do we build the Universal Turing Machine that does what is required?

Step 1: Encoding a Turing Machine

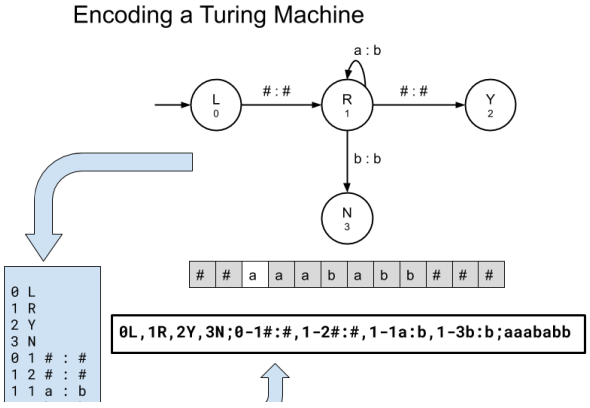
To encode a Turing Machine completely, we must capture several pieces of information.

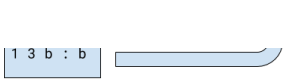
- First we enumerate all of the states.In addition to labeling the states with "R", "L", "Y", etc. we also give each state a unique number. We use 0 to represent the initial state.
- Next we list every state transition. We describe where the arrow begins and ends with state numbers. In addition to that we have to describe the read : write instruction on the arrow.
- Finally, we can also include the Turing Machine's own tape by simply appending it to the end.

This results in a single string of symbols that describes a Turing Machine and its input.

Study the example on the next slide to see how this is done.

Step 1: Encoding a Turing Machine





The result: "
0L,1R,2Y,3N;0-1#:#,1-2#:#,1-1a:b,1-3b:b;aaababb" tells us everything we needed to know about that Turing Machine. If we needed to, we could redraw the diagram just from this encoding.

Though, as mentioned earlier, Alan Turing had a unique motivation for encoding a TM as a data string. He wanted to place this data on the input tape of another Turing Machine.

He saw that it was possible to design a Universal Turing Machine that could take in one of these encodings and exactly simulate what it would do as it processed its data.

Step 2: Build a Universal Turing Machine

The UTM sounds too good to be true. It can simulate the behavior of any individual TM. In modern terms, the UTM is an *interpreter* - it performs a step-by-step simulation of any Turing machine. Program and data are the same thing - a program (Turing machine) is nothing more than a sequence of symbols that looks like any other data.

It is beyond the scope of what we can do here to describe a specific implementation of a UTM. Building a UTM appears to be a daunting task. Alan Turing described the first such Turing machine in his 1937 paper. In 1962, Marvin Minsky discovered a remarkable 7-state UTM that executes using a four-symbol alphabet, but it is rather complicated to describe. As recently as 1996, one was discovered with only 4 states and 6 symbols.

1. **Short Answer Response**

   Write one short paragraph (4-8 sentences) that answers the following questions: What makes the Universal Turing Machine different from any Turing Machine we have seen so far? How are the details of a Turing Machine able to be encoded on a memory tape? (1 pt)