A Guided Tour to Turing Completeness

Benji Altman

May 18, 2018

Abstract

The common consensus within the mathematical and computational communities is that there are no models for computation that can do something a Turing machine can not. In this paper we look at different models of computation that are equally powerful to a Turing machine and ones that aren't as powerful in an attempt to find some of the fundamental building blocks that make something Turing complete.

1 Introduction

In computer science we have an idea called Turing Completeness. In order to fully understand this we first need to examine what a Turing machine is. Our goal however isn't to just explain Turing Completeness, but rather to understand what the building blocks of a Turing complete system is, and in doing this we will want to explore multiple Turing Complete systems and some systems that are not quite Turing Complete in order explore their similarities and differences.

1.1 Terminology

Before we can get started we need to define the notion of a Language.

Let Σ be a finite set, we may call it our alphabet. We say x is a character or x is a symbol if $x \in \Sigma$.[1] A string on Σ is defined as a finite sequence of symbols. Finally A language L on Σ is defined to be a set of strings.

For notational reasons we write strings somewhat differently then we usually would a finite sequence. For example, let $\Sigma = \{1, 2, 3\}$, then 1123 is the same as the finite sequence $(1, 1, 2, 3)$. This can be confusing if for example $1123 \in \Sigma$. To avoid this issue we will write $1123 = (1, 1, 2, 3)$ and $\{1123\}$ is the string with just the single element 1123. In this paper we will also never have a symbol that could lead to any ambiguity. It is also worth noting that the string with no elements exists. We will, unless otherwise stated, denote this with λ; this will also be called the empty string.

Finally, if s is a string we may refer to it's first character as s_0 and it's second as s_1 and so on. For the entirety of this paper all counting starts at 0 as it is considered a natural number.[2]

2 Deterministic Finite Automata

The first machine we investigate is a Deterministic Finite Automata, or DFA for short. A DFA has a set of states, a set of instructions. One state must be a special start state and at least one state must be an acceptance state. Instructions have three parts: a symbol, a start state, and an end state. When the DFA is given a string it starts at it's start state, q_0. If there is an instruction that has q_0 as the start state and whatever the first character in the string is, then we go to the end state given in the instruction and start again with the first character of the string removed and starting in our new state. If there is no instruction then the DFA rejects the string.

For example, if I am given the string aab and my DFA has states $\{q_0, q_1\}$ with q_0 being the start state and q_1 being the only acceptance state. Additionally it has instructions:

[1] symbol and character are used interchangeably in this paper and have the exact same meaning unless otherwise denoted.

[2] `https://www.cs.utexas.edu/users/EWD/transcriptions/EWD08xx/EWD831.html`

#	Start State	Character	End State
0	q_0	a	q_0
1	q_0	b	q_1
2	q_1	b	q_1

Then the DFA works as follows

1. Start at q_0 (the start state) with string aab. a is the character so we use instruction 0 and go to state q_0.
2. We now are at q_0 and we have the string ab and we again go to q_1 by instruction 0.
3. We now are at q_0 and we have the string b and we now use instruction 1 to go to q_1.
4. We now have an empty string, we are on state q_1 which is an acceptance state so this string is accepted.

Every DFA defines a language. The language it defines is the set of strings that the DFA accepts. Any language that has a DFA is called a regular language. For example any finite language (a language with only finitely many strings) is regular as we may make a DFA that accepts every individual string in the language. The language $\{\lambda, a, aa, aaa \ldots .\}$ is regular as we will have only a single state q_0 that is both an acceptance state and the start state and the only instruction will be q_0 goes to q_0 when an a is seen.

There is a corresponding notation that actually makes all this quite easy, called regular expressions. There are two kinds of regular expression: there is the formal mathematical regular expression that we will explore momentarily, and the computer programming regular expression that is based of the former, however is far more complicated.

We define a regular expression to be read as follows. You may have a single symbol a, this is a valid regular expression and represents the language containing only a. If p and q are regular expressions then p

- If a is a symbol, then a is a valid regular expression representing the language $\{a\}$. That is the language with only a the string a.
- Recalling that λ represents the empty string, λ is a valid regular expression representing the language $\{\lambda\}$, that is the language with only the empty string.
- Let p and q be valid regular expressions with p representing languages P and Q respectively. We say $(p) + (q)$ is a valid regular expression representing the language $P \cup Q$.
- Let p and q be valid regular expressions with p representing languages P and Q respectively. We say $(p)(q)$ is a valid regular expression. In order to define what it represents we will adopt the notation that for strings a and b, ab is a a concatenated with b. Now the language defined by $(p)(q)$ will be $\{ab \,|\, a \in P \wedge b \in Q\}$.
- Let p be a valid regular expression representing language P. We say $(p)^*$ is a valid regular expression representing language K, which we define recursively as $K = \{\lambda\} \cup \{ab \,|\, a \in K \wedge b \in P\}$.

We now make it easier to write by taking out the necessity for all the parentheses by adding an order of operations.

1. The Kleene star: a^*
2. Juxtaposition: ab
3. Addition: $a + b$

This means the DFA we gave as an example above, may also be represented as a^*b^*, why this is left to the reader as a fun exercise.

3 Push Down Automata

Now it's worth noting that many every day languages are regular. For example, telephone numbers or email addresses. However the language of all valid regular expressions is not itself regular. The reason for this is the matching of parentheses. Take a simpler example, the language $a^n b^n$. This would be the language $\{\lambda, ab, aabb, aaabbb, \ldots\}$. Now a regular expression has no idea of memory so this language must not be regular. Here we introduce a stack.

A stack is an object that allows for two fundamental operations. Push and Pop. If we have a stack S and we push a onto S then a has been added to the top of S. When we Pop from S whatever is on top of S is removed and returned. This means to see what is at the bottom of a stack, or indeed how many items are in a stack, one must throw away all the items. This is a very limited form of memory, however with it we can now tackle problems like $a^n b^n$ or even the language of all valid regular expressions.

We construct a Push Down Automata, or PDA for short, from a DFA. We add to our DFA a stack, and every instruction may be based on what is on top the stack and also may give us stack instructions. The stack instructions must be a finite list of pops or pushes. The stack starts out empty, and it may contains only symbols.

This defines a new class of language, a Context Free Grammar. Now any Regular Language is a Context Free Grammar as the PDA could just ignore it's stack, however let us construct a PDA for $a^n b^n$ to show we can do something new. Let our state set be $\{q_0, q_1, q_2, q_3\}$ with q_0 and q_3 being acceptance states and q_0 the start state. We also allow λ to represent what is read when the stack is empty, this will be important.

#	Start State	Character	Top of Stack	End State	Stack Operations
0	q_0	a	λ	q_1	
1	q_1	a	λ	q_1	**Push**(a)
2	q_1	a	a	q_1	**Push**(a)
3	q_1	b	λ	q_3	
4	q_1	b	a	q_2	**Pop**
5	q_2	b	a	q_2	**Pop**
6	q_2	b	λ	q_3	

In order to test this let try out this machine on the string $aabb$.

1. We start with the string $aabb$ and at state q_0, and our stack is empty. As such we use instruction 0, and we simply go to state q_1.

2. We are at state q_1 with an empty stack and string abb. We then follow instruction 1, staying at state q_1 and pushing an a onto the stack.

3. We now have the string bb, while at state q_1 and having stack a. Due to this we follow instruction 4, go to state q_2 and pop off the top of the stack.

4. We now have the string b, with empty stack and at state q_2. We must now follow instruction 6 and go to state q_3. We have now gone through the string and have ended on an acceptance state.

Now, Push Down Automata are much more powerful then a regular language. There is a lot that can be done with just a stack. For example, given a specific alphabet Σ, the language of all regular expressions on that alphabet has a push down automaton:

#	Start State	Character	Top of Stack	End State	Stack Operations
0	q_0	(	λ	q_0	**Push**((
1	q_0	$\sigma \in \Sigma$	λ	q_1	
2	q_0	(	(	q_0	**Push**()
3	q_0	$\sigma \in \Sigma$	(	q_2	
4	q_0	(	)	q_0	**Push**()
5	q_0	$\sigma \in \Sigma$	)	q_0	
6	q_1	+	λ	q_0	
7	q_1	*	λ	q_1	
8	q_1	$\sigma \in \Sigma$	λ	q_1	
9	q_1	(	λ	q_2	**Push**((
10	q_2	+	(	q_0	
11	q_2	*	(	q_2	
12	q_2	$\sigma \in \Sigma$	(	q_2	
13	q_2	(	(	q_2	**Push**()
14	q_2	)	(	q_1	**Pop**
15	q_2	+	)	q_0	
16	q_2	*	)	q_2	
17	q_2	$\sigma \in \Sigma$	)	q_2	
18	q_2	(	)	q_2	**Push**()
19	q_2	)	)	q_2	**Pop**
20	q_0	λ	λ	q_1	
21	q_0	λ	(	q_2	
22	q_0	λ	)	q_2	
23	q_1	λ	λ	q_1	
24	q_2	λ	(	q_2	
25	q_2	λ	)	q_2	

[3] Where q_0 is the start state and q_1 is the only acceptance state.

Now one might logically ask, what happens if we add another stack to this machine, or change it from a stack to a queue[4]. In either case we actually end up with something that is equivalent to a Turing machine.

4 Turing Machine

A Turing machine has a 'tape' that is infinitely long in both directions. Each spot on the tape can either be empty, or may contain a symbol. We start out with the tape empty, except that the string that we operate on is written onto the tape. For example if we were operating on *abc* then the tape would be $(\ldots, \lambda, \lambda, a, b, c, \lambda, \lambda, \ldots)$. The Turing machine has a head (much like a typewriter or 3D printer) which can move along the tape. The head starts at the beginning of the input string (in our example it would start at the spot with a). Now each machine has a set of states, much like before. The only difference with our states is that we do not have acceptance states, rather this a a single Halt state that ends the machine. To test for acceptance we print a response onto the tape and leave the head at the beginning of the response. Our instructions will be similar to before, with the components being: Start State, Character at Head, End State, Character to Write, Direction to Move. The start and end state are just like before, with the special condition that end state may be the special Halt. The character to read and write may be any character or an empty (λ). The direction to move may either be left, right, or stay; this will tell the head where to move on the tape. If at any point in the execution of the program, there is no instruction that is valid, halt immediately.

Turing machines aren't simple and even their smaller examples can still be quite large. The special about a Turing machine is that it can validate nearly any language you can think of. The set of languages it can validate are called computable. Additionally the Turing Machine can do much more then that. It can

[3]Here where we expect the character lambda (instructions 20-25), we actually mean to use lambda as a symbol as it is needed in regular expressions, however when we say top of stack has a lambda, that is the stack is empty.

[4]A queue is like a stack, but symbols are added at the bottom instead of at the top.

give outputs that aren't just true or false. Simply by writing something the tape it can give any output at all. You can construct arithmetic on anything you have a way to represent within the Tape using a Turing Machine. This is most rigorously put in the Church Turing Theses

> "A function on the natural numbers is computable by a human being following an algorithm, ignoring resource limitations, if and only if it is computable by a Turing machine."
> — Wikipedia

This is saying that any algorithm you can do can be done by a Turing Machine. Now whats remarkable about this statement is that despite a lack of proof, it has been accepted nearly universally by academia. So far there has been no reason to doubt this. Here is where the term Turing Complete comes in.

Definition 4.1. A system is Turing Complete if it can simulate any Turing Machine.

For example, the system called Turing Machine is Turing Complete (trivially so), however a specific Turing Machine may not be, as it can only do one task. There are however Universal Turing Machines that take in another Turing Machine (formated as a string in some way), followed by some input, and can simulate the Turing Machine on that input. These Universal Turing Machines are themselves also Turing Complete.

5 Other Turing Complete Systems

Turing Completeness is the highest level of computable power that a system can have, despite this Turing Complete systems pop up in games and other places by accident all the time. For example Magic the Gathering, a popular card game, can implement any Turing Machine with up to a certain number of states on a alphabet with a fairly small number of characters. There is a Universal Turing machine that fits within these requirements and as such Magic the Gathering is Turing Complete. We won't be looking at Magic's Turing Machine as it's convoluted and uninteresting, it simply is an example of how even in unexpected places Turing Completeness pops up. Another good example of this is C++, a programming language. C++ was by all means meant to be Turing Complete, and it is, however within C++ there is a system called Templates. This template system is not so much part of the executing code and is rather just things that are done as the code is turned from C++ into Computer readable binaries. This system is to help with some concepts like abstraction in C++, however it was found that Templates, completely by accident, were also Turing Complete. Rather than looking at these complicated examples, lets look at some simple systems that are Turing Complete.

5.1 Procedural Languages

Procedural (or Imperative) programming Languages are the most standard type of programming language. They tend to achieve Turing Completeness by the use of a few simple components

- **Steps** - A program's most basic feature is stepping through. First line 1 happens, then line 2, and so on. This is very similar how states work in a Turing Machine.
- **Variables** - Variables are a way of holding data that can be changed, this is like the tape in a Turing Machine. Notice it is different then the stack from a Push Down Automata as it can access any information without throwing out data.
- **Loops and Conditionals** - A loop allows one to keep doing the same steps over and over again. This way a program doesn't just run through each step and then end. A loop also must have some way to end on some condition. In it's most basic form these work by having a conditional 'jump' statement. It's allows you to go to any line of code on some condition. If the steps in a program are like the Turing Machines' states, then these are like the instructions allowing us to go where we want based on what is in the tape (or variables in this case).

Common languages of this type are: Java, C++, C, JavaScript, Python, R, Assembly Languages, Fortran, Basic.

5.2 Functional Languages

Where procedural languages work by stepping through a program, functional languages are completely different. They simply use Recursion to recreate all the things that a Procedural language can do. The basic idea here is that we can get looping by using recursion, and we can also use recursion to create new data, so we don't need loops or variables. Each function call creates new constant values when it is called. We still have conditionals as we will need to have end conditions in our recursion. This is actually how the Templates in C++ get their Turing Completeness.

Common languages of this type are: LISP (and it's descendent's), Haskell, F#, and Mathematica.

In reality many procedural languages support functional programming and vice versa. For example C++, JavaScript, Python and R can all be functional.

5.3 Other

There are other weirder and less common ways to achieve Turing Completeness that are actually used in programming languages. I would rather focus on simple systems that are Turing complete, however not programming languages. The simplest system that one can formalize and find to be Turing Complete are Cellular Automata. John Conway's Game of Life is the most famous and with only a few simple rules on an infinite 2-D board, the Game of Life is Turing Complete. Even more surprising is the Automata Rule 110. This Automata is much like Conway's Game of Life but on a 1-D board, and with even simpler rules.

6 Conclusion

Now we've seen some ways to create Turing Completeness and some methods that don't quite make it, it's hard to say what exactly is needed to get Turing Completeness, but there does seem to be some common themes. The ability to store data and act on any part of that data without having to throw out information seems to be the key. For example in a Push Down Automata with a Queue instead of a Stack, one can simply cycle through the stack as much as they need and find any of the data.[5] It's hard to say if this by itself is enough, and as far as I can tell, there is no formal way to tell if something is Turing Complete other than just implementing a Universal Turing machine in it, but this does seem like a good litmus test. It would be nice to look at more systems that are weaker than a Turing Machine in order to get more contrast.

[5] In the two stack system you can either simulate a queue or go straight to simulating a Turing Machine by treating one stack as the tape to the left of the head and the other stack as what the head is pointing to and everything to the right of that.