Languages (Turing Machines)

Carol Zander

Language history

While we tend to build up from regular languages, in fact, the development occurred in a different order.
1930-40s: The Turing machine (most powerful automaton) was developed (the Post machine, equivalent to the Turing machine was developed at the same time).
1950s: Regular expressions and Deterministic Finite Automata was developed.
1960s: Context-free grammars and pushdown automaton was developed.

Turing machines (TM)

Turing machines are named after Alan Turing (1912-1954), who invented them. Although he died in 1954, he is one of the most famous computer scientists. Among his accomplishments:

- Inventing the Turing machine
- Influential in early artificial intelligence (AI) (including the Turing test), known as the father of AI
- During WWII, was instrumental in a device to break the code of the Enigma machine
- Created one of the first designs for a stored program computer
- The Turing Award (like a Nobel prize for computer scientist) was named after him

He committed suicide at 41, possibly due to hormone treatments he was forced to take owing to a conviction for gross indecency (arrested in 1952, it was illegal to be a homosexual at that time).

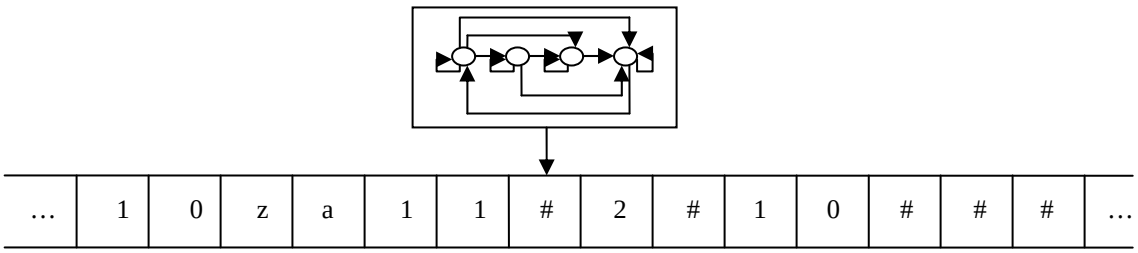
Finite state automata can be used to recognize only regular languages. To recognize a context-free grammar, we need to use a **pushdown automaton**. This is a finite state automaton that has an infinite stack that provides memory. If we want to recognize a context-sensitive grammar, we need to use a **linear bounded automaton**. These have a tape upon which they can read/write symbols (but they are restricted from complete use of it). To recognize recursively enumerable languages (phrase-structure grammars), we need a **Turing machine**, which has a tape and unrestricted use of it. In fact, Turing machines are very important theoretical constructs in computer science since they (despite seeming simplicity) can perform any operation that any modern-day computer can.

The Turing machine is a theoretical model of computing that is similar to a finite state machine, but more powerful. Although it seems primitive, it is able to **perform every operation that a modern computer can perform.** It is still heavily studied as a general model of computation. In addition, to having states (like an FSM), a Turing machine has one-dimensional tape that it can write on that is infinitely long. A tape head points at the current cell to be read. At each step, the Turing machine reads a character from the tape, writes a character back to the tape, and moves either left or right. This is sufficient to implement every algorithm that we have studied (and any other algorithm). A Turing machine can be defined by:

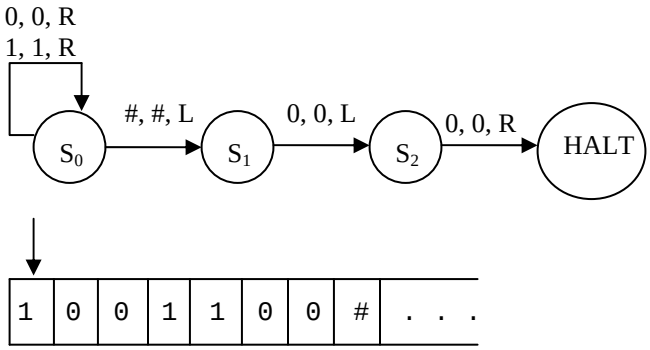
- S: finite set of states
- I, O: alphabets for input/output (including the blank symbol '#')
- S_0: start state
- f: function that is given the current state and the character that is read from the tape, the function yields a new state, an output character, and a direction to move. This function does not need to provide a new state for every pair of a state and input letter. If no new state is specified, then the machine *crashes* for the state/input pair.

  The function statements are, for example: (a, b, R)
  The first character is the input to be read, the second character is the output to write, and the last is the direction to move (either L for left, or R for right).

This picture illustrates a Turing machine with a particular position with respect to the tape:



Although we can perform any algorithm, complex operations can be very tedious to specify on a Turing machine. To recognize a string in the language, the concept of a final state, usually called a *HALT* state, is used. A Turing machine recognizes a string if it finishes in a HALT state. Start with a relatively simple Turing machine (since the language is regular). One Turing machine to recognize all strings that end in 00 looks similar to a finite automaton. It is typically assumed that the Turing machine starts at the left-most non-blank symbol on the tape and everything after the input is blank.



The Halting Problem

One reason that Turing machines are useful is to prove things about computation (for example, proving that certain problems cannot be solved). The most famous example is the halting problem. The question is whether it is possible to decide whether an arbitrary Turing machine T halts on an arbitrary input w. (Otherwise, it crashes or runs infinitely.) This question is applicable more generally to any programming language. Can you determine for all programs and all inputs whether the program halts on the input?

Halting Problem — Given some arbitrary Turing machine, called T, and some arbitrary input, called w, is there an algorithm to decide whether T halts when given input w?

At first reading, this may seem like gibberish, but think of T as your program, w as the data to your program (w is for word), and an algorithm as just another program (another Turing machine). This should be familiar to you; a compiler (a program) takes your program as input and performs a task. So, the Halting problem says, given a program with data, can you write another program to detect if your program normally terminates? Or, another way of thinking of it, could you detect an infinite loop by just examining the program and data?

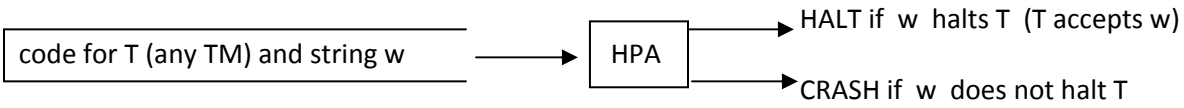
Clearly, in some cases, this can be decided, but the halting problem asks whether this can *always* be decided. Unfortunately, the answer is no. This is an example of an unsolvable problem. Thus, we get a well known result:

Theorem: The Halting Problem is not solvable.

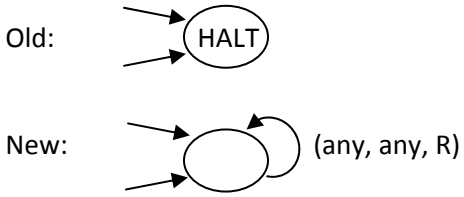
The proof of this is interesting. It is a proof by contradiction. Since a Turing machine can solve any solvable problem, we could build a Turing machine to solve this problem if it is solvable. To accomplish this, we encode the input Turing machine as a string and place it on the tape, with the input to the encoded Turing machine on the tape afterwards.

Let's assume that we have a Turing machine that can solve the halting problem - T_H. We now construct a Turing machine T_H' that halts if T_H says that the program would not halt and runs infinitely if T_H says the program would halt. Since T_H' can take any Turing machine as input, what happens when T_H' is given T_H' as both inputs? That is, it needs to decide if T_H' halts with T_H' as its input. If T_H' halts that implies that T_H' doesn't halt (by nature of the construction). If T_H' doesn't halt, it implies that T_H' halts. This is a contradiction and so there can be no such T_H' or T_H.

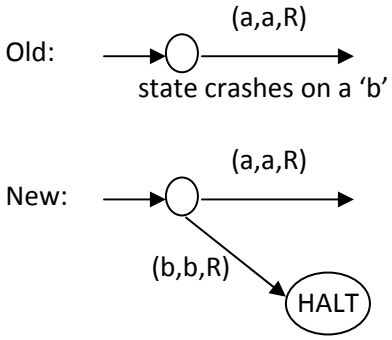
Proof: By contradiction.
Suppose there exists some TM called HPA (for Halting Problem Answerer) that takes any T and any w and halts and answers yes or no (we could write special symbols representing yes and no). Note that HPA does not loop.



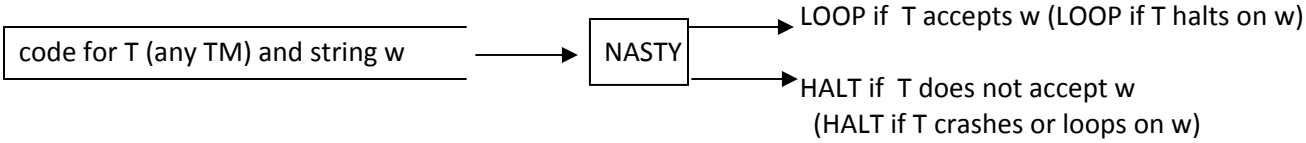
Now make another TM that is similar to HPA, but change the HALT state to an infinite loop. For example,



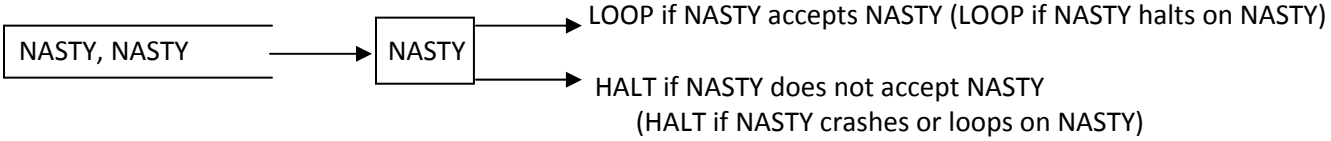
Also change the CRASH states to HALT states. For example,



Call this new Turing machine, NASTY. This machine is equivalent to HPA except that CRASH and HALT states have switched roles. So, if we draw this new machine,



Just as any T and any w can be fed into HPA, any T and any w can be fed into NASTY. So give it both NASTY as T and NASTY as w. NASTY is also encoded and on the tape as the machine dealing with T and w. Plug in NASTY for T and w in the above picture.



This is a contradiction. It says that NASTY loops if it halts and it halts if it loops. Since NASTY is equivalent to HPA, HPA cannot exist. Therefore there is no Turing machine that solves the halting problem.