Logic for Linguists: Lecture 6

Gregory Wilsenach

University of Cambridge

20th November 2019

Review

Last week we discussed the Chomsky Hierarchy.

- Type-0: Recursively Enumerable Languages,
- Type-1: Context Sensitive Languages,
- Type-2: Context-Free Languages, and
- Type-3: Regular Languages.

We discussed regular languages and gave a number of characterisations of these languages.

Algorithms and Regular Languages

We can define regular languages three different ways, via:

- Finite State Automata,
- Regular Construction, and
- Regular Grammars.

We can similarly define context-free languages as either those accepted by a (non-deterministic) push-down automata or those definable by a context-free grammar.

In fact, each class in the hierarchy is characterised both by a formal grammar and by a machine model.

These are all algorithmic notions!

Limits of General Algorithms?

In this respect algorithmic notions and formal linguistic notions (grammars) correspond.

Chomsky's goal was to develop grammars (algorithmic method of specifying a language) which were general enough to encode the syntactic structure of natural language. In other words, they could decide if a given string is a well-formed sentence of a particular natural language.

We also know that regular languages are very limited, and certainly fail to meet this lofty goal. It's possible to show via very similar means that context-free and context-sensitive languages also fail in this respect.

In this lecture we will introduce arguably the most general algorithmic notion and define recursively enumerable languages, the top of the Chomsky Hierarchy.

Examples of Algorithms

Let's think about algorithms.

We have algorithms that solve many natural problems. For example:

- multiplication of two numbers,
- addition of two numbers,
- checking if a quadratic formula has real solution, and
- checking if a given string is a member of some regular language.

Example

Let's think through the algorithm for checking if a given string x is a member of a regular language L.

We know that since L is regular there is a DFA that decides it. We then

1. start at the starting state;
2. check if x is empty, if it is and we are in a final state then output that $x \in L$ otherwise output that $x \notin L$;
3. take the first symbol s in x and see if there is an arrow leaving the state labelled by s;
4. if there is no such arrow, then output that $x \notin L$ otherwise transition along that arrow to the new state; and
5. remove the left most element of x and go back to step two.

So what is an Algorithm?

We can extract some common features of an algorithm from our examples. They

- are finitely describable processes,
- are defined in terms of elementary operations (which are "local"),
- can go on forever or stop and produce some result.

We say an algorithm solves some problem (say, multiplication of two numbers) if when given an input it

- always produces the correct answer and
- stops running at some finite time in the future.

Hilbert's Entscheidungsproblem

The modern history of the theory of algorithms begins with an interesting philosophical question: Is there an algorithm for truth?

More formally: Is there an algorithm which takes in a statement about arithmetic (say in first-order logic) and works out whether that statement is provable from the axioms of arithmetic.

This question was posed by Hilbert at the 1928 International Congress of Mathematicians, and he believed the answer was a resounding yes.

Today, we will prove him wrong.

What is an Algorithm *really*?

Hilbert had no formal notion an algorithm, and that's not a problem if you think the answer is yes.

We can mostly recognise an algorithm when we see one, and so to answer the question in the positive you just need to present an algorithm that provably solves the problem.

However, to answer it in the negative you need a formal definition. We need some formal definition of what it means to be an algorithm so that we might prove that no such thing can solve this problem.

How should formalise this millennia old notion?

Enter Turing and Church

This is the question to which so many great logicians devoted themselves.

In this lecture we will introduce Turing's model of computation, appropriately called the Turing Machine.

We will discuss next week Church's Lambda Calculus, an alternative formalisation with deep ties to other areas in logic (including type theory).

The Lambda Calculus is of particular interest to many people in formal linguistics.

Informal Turing Machines (1)

Let's begin with an informal introduction to this formal notion. Turing started with a simple idea: How can we model a human working out a problem?

The Setting:

- Let us suppose a human is sitting at a desk.
- The human has access to some finite set of symbols (the vocabulary Σ).
- The human has access to an unbounded stack of paper right next to him, where each page has a number starting at 1.
- A piece of paper can hold exactly one symbol.
- The human can be in one from some fixed finite number of mental states at any given time (the set of states Q).

Informal Turing Machines (2)

In each step of computation the human reads the piece of paper in front of him and solely on the basis of his mental state and the symbol he reads he may do one or more of the following:

- he may erase what's on the page and replace it with any symbol (or delete everything on the page),
- he may move forward or backwards a finite number of pages in his stack, or
- he may transition his mental state.

There is a special state (or set of states) called the halting, final, or accepting state, and when he transitions into that state he is done, and the output of his work is what is written on the tape.

Example

Let's suppose we want to design an algorithm that checks if an input is a string of a's of even length.

Let the vocabulary be $\{a, b\}$ and let the set of states be $Q = \{s_0, s_1, \text{YES}, \text{NO}\}$. The states YES and NO are final states. We start in the s_0 state.

The pages are all blank except for the first n of them which each have a symbol on them. We take this to be our input.

We compute as follows:

- If we read an a: We transition to s_1 if we are in state s_0 and transition to state s_0 if we are in state s_1 and shift one page up.
- If we read a b: We transition to NO (which is halting).
- If we read a blank: We transition to YES if we are in state s_0 and transition to NO otherwise.

Formal Turing Machines

A Turing Machine M is a 7-tuple:

$$M := (Q, \Sigma, b, \Gamma, s_0, F, \delta),$$

- Q is a finite non-empty set of states,
- Σ is a finite non-empty set called the vocabulary,
- $b \in \Sigma$,
- $\Gamma \subseteq \Sigma$ is the vocabulary of the input,
- s_0 is the initial state,
- $F \subseteq Q$ is the set of accepting or halting states, and
- $\delta : (Q \setminus F) \times \Sigma \to Q \times \Sigma \times \{L, R, N\}$ is the transition function.

Computation

We compute on an infinitely long tape consisting of individual cells (think of these as the individual pieces of paper).

The tape has something initially written down using the symbols in Γ, this is the input.

We start at the leftmost side of the tape and in the starting state and at each step apply δ.

We note that δ takes in a (non-final) state and the symbol written in the cell we are currently reading and tells us

- what state to transition into,
- what we should write down on the current cell of the tape, and
- whether we should go left L, right R, or go nowhere N.

Decidable Languages

We often think of a special type of Turing machine with two final states one called YES and the other called NO. We call this a decision Turing machine.

Let Γ be a vocabulary. Recall that Γ^* is the set of strings over Γ.

Let $L \subseteq \Gamma^*$ be a language. We say that a decision Turing machine M with input vocabulary Γ decides L if M always halts on any input and for every $x \in \Gamma^*$:

- if $x \in L$ then M halts with YES given input x and
- if $x \notin L$ then M halts with NO given input x.

We say that a language $L \subseteq \Gamma^*$ is decidable if there is a decision Turing machine M that decides L.

Recursively Enumerable Languages

We say that language $L \subseteq \Gamma^*$ is recursively enumerable (RE) if there is a decision Turing machine M such that for every $x \in \Gamma^*$ if $x \in L$ then M halts with YES given input x and otherwise either halts in some in some other state or runs forever.

Examples

Example 1: We have already seen that the language consisting of strings consisting of an even number of a's is a decidable language.

Example 2: The empty language is decidable.

Example 3: The language $\{a^n b^n : n \in \mathbb{N}\}$ is decidable.

In fact, it's possible to show that all context-sensitive, context-free, and regular languages are decidable.

We can in fact show that any known programming language can be compiled down to Turing machines.

Some History

Why not just add something extra to the Turing machine and develop a more powerful notion of an algorithm.

In the years following the development of the Turing machine and the Lambda Calculus hundreds of researchers around the world developed their own notion of an algorithm.

What is truly amazing is that for all of these models it was discovered that if that model can decide a language then there is a Turing machine that can do it as well.

Church-Turing Thesis

This led to the Church-Turing Thesis:

If some problem can be solved by purely mechanical means then it can be solved by a Turing machine.

In other words: The Turing machine is the maximally powerful formalisation of an algorithm.

So what can't we do using algorithms? What languages will we forever be unable to recognise?

The Halting Problem

Is there a decision Turing machine M_H that takes as input (M, x) where M is a description of a Turing machine and x is some input string x and

- halts with YES if when M is run with input x then it halts at some point and
- halts with NO if when M is run with input x it does not halt at some point (it keeps running forever)?

No there is not...

Turing proved that there is no such algorithm. Here is the proof:

Let's suppose that such an M_H exists. Let C be a Turing machine implementing the following algorithm:

"for an input string y, run M_H with input (y, y) and if M_H halts with NO then halt with YES and if M_H halts with YES loop forever."

Now we run C on itself!

Suppose C halts with input C. Then $M_H(C, C)$ must halt with NO in which case C run on input C must fail to halt.

Suppose C does not halt with input C. Then $M_H(C, C)$ must halt with YES. But then C does halt on input C.

We get a contradiction either way!

Poetry Slam Time

A poetic proof of the halting problem. It's wonderful, read it:

- `http://www.lel.ed.ac.uk/~gpullum/loopsnoop.html`

Answering Hilbert

Turing was able to show using this result that Hilbert was wrong, there is no machine that can decide the set of true sentences of arithmetic.

He did it by showing that we can encode the sentence "this machine halts on input x" as a statement about arithmetic.

We have since proved that many problems are undecidable. For example: Given two context-free grammars, do they define the same set of sentences?

Reading

An introduction to formal models of computation and a proof of the halting problem:

- `https://www.cl.cam.ac.uk/teaching/1718/CompTheory/lecture-1.pdf`

A video of a Turing machine in running and a detailed visual description of what a Turing machine is:

- `https://www.youtube.com/watch?v=E3keLeMwfHY`
- `https://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/turing-machine/one.html`

A very detailed philosophical account of Turing machines and the theory of computation:

- `https://plato.stanford.edu/entries/turing-machine`