Unrecognizable and Undecidable Languages

Jeffrey Shallit
School of Computer Science
University of Waterloo
shallit@uwaterloo.ca
https://cs.uwaterloo.ca/~shallit
Today you will see one of the most fundamental and important results in the theory of computation: namely, there exist some natural computational problems for which there is no algorithm.

By considering “membership in a language” as a computational problem, and equating “algorithm” with “always-halting Turing machine”, we translate this to “there exist some languages whose membership problem cannot be solved by any Turing machine”.

In other words, there exist some languages that are not Turing-recognizable, or Turing-decidable.

These results were (essentially) found by Turing in 1936.

It was an intellectual revolution comparable to quantum mechanics in physics.
Recognizable and Decidable Languages

Recall that a language $L$ is *Turing-recognizable* (aka recursively enumerable) if there is a Turing machine $T$ such that $L = L(T)$. 

A recognizer $T$ must halt and accept if $x \in L$, but it is allowed to run forever if $x \notin L$.

And a language $L$ is *Turing-decidable* if there is a Turing machine $T$ that *halts on all inputs* such that $L = L(T)$.

A decider $T$ is guaranteed to halt on every input, by eventually reaching either $q_{\text{acc}}$ or $q_{\text{rej}}$.

In both cases we are working with deterministic TM’s.

If a language is not recognizable, we say it is *unrecognizable*. If it is not decidable, we say it is *undecidable*. 
In this lecture we will routinely be feeding encodings of TM’s into other TM’s, as inputs.

You already know, in your computing career, examples of this kind of thing.

For example, a compiler is a program that takes (for example) a Java program as input and returns Java bytecode or native machine code as output.

We will even feed encodings of Turing machines into themselves.

Do not be scared of this!

For example, a “prettyprinter” is a program that takes a program as input and outputs a nicely formatted version of that program.

Then you could call the prettyprinter on itself.

So there is nothing conceptually new here.
Last time we proved (by a counting argument) that there exists a language that is not Turing-recognizable.

Now let’s provide (by explicitly constructing an example)

- an example of a language that is not Turing-recognizable, and
- an example of a language that is Turing-recognizable but not Turing-decidable.

I will use the notation $e(T)$ to denote the binary encoding of a Turing machine $T$. (Prof. Watrous’s notes use the notation $⟨T⟩$; you can use either one that you like.)

It does not matter which encoding we use, as long as it satisfies the four properties: (every TM has an encoding, one can uniquely decode, it’s possible to check if a purported encoding is valid with a TM, and it’s possible to determine, when two encodings are concatenated, where one encoding ends and the other begins). But once we decide on an encoding, we use the same one all the time.
Here is our example of a language that is not Turing-recognizable.

\[ \text{DIAG} := \{ e(T) : T \text{ is a DTM and } e(T) \not\in L(T) \} . \]

This is the language of all (encodings of) Turing machines that do not accept their own encodings.

We will prove this using an argument analogous to the Cantor diagonal method that we discussed in last week’s lecture.

Let us assume that DIAG is Turing-recognizable.

Then there would be some Turing machine \( M \) that recognizes it. So \( L(M) = \text{DIAG} \).

Now the question is, is \( e(M) \in \text{DIAG} \)?
A language that is not Turing-recognizable

Is

\[ e(M) \in \text{DIAG} = \{ e(T) : T \text{ is a DTM and } e(T) \notin L(T) \} ? \]

If so, then by definition \( e(M) \notin L(M) = \text{DIAG} \), a contradiction.

So \( e(M) \notin \text{DIAG} \). But then from the definition of \( \text{DIAG} \) we have \( e(M) \in \text{DIAG} \), another contradiction.

Either way, this contradicts our assumption that \( \text{DIAG} \) is Turing-recognizable.

So \( \text{DIAG} \) is an explicit example of a language that is not Turing-recognizable.
A language that is Turing-recognizable, but not Turing-decidable

Now let’s find an example of a language that is Turing-recognizable but not Turing-decidable.

Our example will be the language called $A_{DTM}$ in Prof. Watrous’s notes. This is the language recognized by the universal Turing machine: $A_{DTM} = \{ e(T)e(w) : T \text{ accepts } w \}$.

First, why is it Turing-recognizable? It is Turing-recognizable because the universal TM $U$ recognizes it! On input $e(T)e(w)$ the machine $U$ runs $T$ on $w$ and accepts if $T$ accepts $w$.

It’s a little harder to show that $A_{DTM}$ is not Turing-decidable.
A language that is Turing-recognizable, but not Turing-decidable

Assume $A_{DTM} = \{ e(T)e(w) : T \text{ accepts } w \}$ is Turing-decidable, and $T_0$ is a Turing machine that decides it.

Then $T_0$ is guaranteed to halt on all inputs. (Note: if the input to $T_0$ is not of the form $e(T)e(w)$ for some $T$ and $w$, then $T_0$ halts and rejects.)

Now let’s create a new TM $K$, that will behave as follows:

If $K$’s input is of the form $e(M)$ for some TM $M$, then $K$ runs $T_0$ on input $y = e(M)e(e(M))$.

- If $T_0$ accepts $y$, then $K$ rejects its input $e(M)$.
- If $T_0$ rejects $y$, then $K$ accepts its input $e(M)$.
- If $K$’s input is not of the form $e(M)$ for some TM $M$, then $K$ rejects. (We often don’t mention this, because it is not relevant.)
A language that is Turing-recognizable but not Turing-decidable

Here is a diagram of $T_0$.

Here is a diagram of $K$.

If $T_0$ existed, we could create $K$ from it.
A language that is Turing-recognizable but not Turing-decidable

Now how does $K$ behave on input $e(K)$, i.e., if we take the parameter $M$ to be $K$?

Well, $K$ runs $T_0$ on input $e(K)e(e(K))$.

Since $T_0$ decides $A_{D_{TM}}$, if $K$ accepts $e(K)$, then $T_0$ halts and accepts. So by definition $K$ rejects its input $e(K)$. A contradiction.

If $K$ rejects $e(K)$ (the only other possibility!) then $T_0$ halts and rejects. So by definition $K$ accepts its input $e(K)$. Another contradiction!

So our assumption that $A_{D_{TM}}$ is Turing-decidable must be false!

This was Turing’s brilliant idea.
A language that is Turing-recognizable but not Turing-decidable

Some people are disturbed by this proof and a tiny minority rejects its validity.

There is even a CS professor at the University of Toronto who claims it is invalid.

However, these doubters are wrong.

The proof has even been validated by proof checkers, so it is simply not in doubt.
Now let’s see yet another example of a problem that is Turing-recognizable but not Turing-decidable: the halting problem.

This is the “holy grail” of software engineering: given a program, we want to know before we run it whether it has an infinite loop in it or not.

Unfortunately, as we will see, there is no computable solution to this problem. So there no “magic bullet” to solve basic software engineering problems.
HALT, the halting language, is defined to be
\{ e(M) e(w) : \text{DTM } M \text{ halts on input } w \}.

To see that HALT is Turing-recognizable, just use a modified version
\( U' \) of the universal TM \( U \). On input \( e(M) e(w) \) what will \( U' \) do? It
will simulate \( M \) on \( w \), and if \( M \) ever halts, \( U' \) accepts and halts.

In other words, if during its simulation of \( M \), the machine \( U' \)
detects that \( M \) enters either the state \( q_{\text{acc}} \) or \( q_{\text{rej}} \), then \( U' \) accepts
and halts.
To see that HALT is not Turing-decidable, we will use the same idea as before.

Assume HALT is decided by some TM, say $T_H$.

Then create a new TM $C$ as follows:

On input $e(M)$, $C$ calls $T_H$ on $e(M)e(e(M))$.

- If $T_H$ accepts, then $C$ runs forever (by entering some infinite loop, for example, by always moving right).
- If $T_H$ rejects, then $C$ halts and accepts.
The halting problem: a picture

Now what happens when we run the TM $C$ with $e(C)$ as an input?
The halting problem

When we run $C$ with $e(C)$ as input, it calls $T_H$ on $e(C)e(e(C))$. So if $T_H$ tells us that $C$ halts on $e(C)$, then $C$ runs forever on $e(C)$.

If $T_H$ tells us that $C$ doesn’t halt on $e(C)$, then $C$ halts on $e(C)$.

We get a contradiction in both cases.

So, in general, it is not decidable if a Turing machine $M$ halts on input $w$. 
The halting problem

Let’s try to understand what this means.

It does *not* mean that for a specific TM $M$ and a specific input $w$, we can never decide if $M$ halts on $w$. In fact, we can often do this! We do it all the time when we are trying to construct a TM with a certain behavior.

It means there is *no general method* that is guaranteed to work for *all* TM’s and *all* inputs. Every finite algorithm (that is, deciding TM) that we produce is bound to fail on *some* input.
More generally

You will notice that our proof for HALT was almost the same as our proof for $A_{DTM}$.

This might make you wonder whether these are both examples of some more general proof technique.

The answer is yes, and is encapsulated in the notion of “reduction”, which we will see in another lecture.