Computability is a robust, stable, and widely-understood concept

Most computer scientists and programmers understand the basic idea of computability. Most programming languages permit expression any computable function, although not necessarily all with the same degree of ease or fluency.

Computability is about algorithms, rather than just Turing machines

The Turing machine is just one of many models for computability, each of which can be mapped to the other. It is a simple base-line model, which can be invoked in the absence of a need for other models. In this note, when we say “machine”, we will mean a Turing machine as a reference model, although it is usually the case that any other equivalent model will work just as well.

Turing’s Thesis

Turing’s thesis is that any computable function can be computed by some Turing machine. The typical invocation of Turing’s thesis is that giving an informal algorithm for computing a function establishes it as computable by a Turing machine, without going into the fine details of how that computation is done on the actual machine. For example, if you give an algorithm for searching a finite directed graph, you don’t have to show explicitly how to map that to a Turing machine. It is understand that you could, by Turing’s thesis. Obviously this is a great time-saver in giving proofs. This thesis is often called the Church-Turing Thesis.

Both programs and data are “data”

A fundamental concept of computability is the ability to treat programs as data. For example, a Turing machine’s program is representable as a string over a finite alphabet. This gives us a needed capability for programs that can inspect and construct other programs.

Universal Computers

There are universal computers, e.g. universal Turing machines. These expect as input an encoding of any program and an input to the program as a single data string. The universal machine can carry out the computation the same as a custom machine based on the program can, by simulating the original program step-by-step. The universal machine may require more steps than the original program would, because it has to interpret the program, as well as carry out the computation. The universal machine could even simulate itself. An easy to see example is the construction of a meta-interpreter for Lisp/Scheme/Racket in its own language.
Simulation

Simulation is the ability to execute a machine computation step-by-step, stopping it in a given state and restarting if desired. Simulation can also involve counting steps of the simulated machine. These aspects can be incorporated into a universal machine to enable proofs that would not otherwise be possible.

Multiplexing, Dovetailing

Certain constructions require running two or more machines simultaneously or in an interleaved fashion (e.g., a step of one then a step of the other). A simulating machine can keep track of the states of arbitrarily-many machines being simulated and can control their stepping. See for example the Complementarity Theorem below.

Encodings of machines

Following Sipser, if M is a machine, then <M> denotes an encoding of M in some fixed alphabet. If x is an input to M, then an encoding of the pair is represented by <M, x>. The details of the encoding are typically not shown, although they could be. For example, the purpose of a Universal machine is to input <M, x> and carry out the computation of M on x.

A machine can examine its own encoding as input. For example, consider a machine M that inputs encodings of machines <N> and determines whether N has more than 100 control states. M could input its own encoding <M> and provide an answer. There are many other day-to-day examples of machines that can input their own encoding. A program that formats programs for printing is one, as is a compiler, which generates machine code from source code.

Strings that aren’t encodings

If we encode machines in an alphabet Σ, some strings in Σ* may be syntactically well-formed and others not. A convenience for avoiding qualifying strings as representing a machine or not is to regard ones that are otherwise not well-formed as representing some standard machine, such a machine that does nothing (immediately stops in the initial state, say). In this way, every string in Σ* represents some machine.

Halting States

A state of a machine is halting if there is no transition from it. It is conceivable that a given control state could be part of a halting state or not, depending on the symbol under the head. For example, there might be a transition defined for δ(q, a) but not for δ(q, b), where q is a control state and a and b are tape symbols. This situation can
be regularized by requiring that if \( \delta(q, a) \) is defined for one symbol then it is defined for all symbols, in which case the only halting states are those \( q \) having \( \delta(q, a) \) undefined for all \( a \). We don’t always bother to regularize the states in this way.

**Acceptance, Rejection, and Divergence**

Halting state can be divided into *accepting* vs. *rejecting*. In order accept a string, the machine halts in an accepting state when started with the string on the tape in the initial state. If the machine instead halts in a rejecting state, it explicitly rejects the string. A third possibility is that the machine never reaches a halting state. In this case we say it *diverges*. It neither accepts nor rejects the string.

**Languages vs. Functions**

A machine can decide a language or compute a function. Deciding a language is a special case of a computing a function with a domain of \( \{0, 1\} \), where a result of 1 means accept and 0 means reject.

**“Problems” as Languages: Solvable vs. Unsolvable**

We often see reference to various problems, such as *The Halting Problem, The Post Correspondence Problem*, etc. These problems are really questions about existence of algorithms, not single puzzles that have a yes-no answer. The algorithm in question is to determine whether a given string, called an *instance* of the problem, is in the language of all strings for which the answer is yes. If there is an algorithm, the problem is called *solvable*. Otherwise it is *unsolvable*.

**Post Correspondence Problem (PCP)**

This is a problem first articulated by Emil Post (1946): Devise an algorithm that will determine from a finite set of pairs of strings \( \{(x_i, y_i)\} \) whether or not there is a sequence of indices \(<i_1, i_2, \ldots, i_n>\), *possibly with repetition of some indices*, such that \( x_{i_1} x_{i_2} \ldots x_{i_n} = y_{i_1} y_{i_2} \ldots y_{i_n} \). A given set of pairs is an *instance* of the problem. The problem itself is that of devising an algorithm.

Computability is *not* about single instances. Here is an instance of the PCP: \( \{(100, 1), (0, 100), (1, 00)\} \). This instance has a solution: the sequence of indices \(<1, 3, 1, 3, 2, 2>\), because (blanks are ignored):

\[
x_1 \ x_3 \ x_1 \ x_1 \ x_3 \ x_2 \ x_2 = 100 \ 1 \ 100 \ 100 \ 1 \ 0 \ 0 = 1 \ 00 \ 1 \ 100 \ 100 \ 100 = y_1 \ y_3 \ y_1 \ y_1 \ y_3 \ y_2 \ y_2
\]

Expressed as a *language*, the language for this problem would be the set of sets of pairs that have a solution. Thus \( (100, 1), (0, 100), (1, 00) \) would be in the language, whereas an instance such as \( (11, 0)(0, 01) \) would not be in the language (check it). PCP happens to be unsolvable.
Decide vs. Recognize

An important distinction arises for languages at the level of Turing machines: A machine can *decide* a language \( L \) if, for a given input \( x \), the machine will eventually terminate with an indication of whether or not \( x \in L \). A machine that decides a language is called a *decider* for the language.

A machine can *recognize* a language if, for a given input \( x \), if \( x \in L \) then the machine will eventually terminate and indicate that \( x \) is in the language, whereas if \( x \notin L \) the machine may or may not terminate.

We will avoid using the term “accept” for a language. Individual strings are accepted, but languages are either *recognized* or *decided* by a given Turing machine.

If \( M \) is a machine, \( L(M) \) is used to denote the set of strings accepted by \( M \), i.e. the *language recognized* by \( M \). \( L(M) \) is also decided by \( M \) provided that \( M \) always halts.

\[
L(M) = \{ x \mid M \text{ started with input } x \text{ eventually stops in an accepting state} \}
\]

Not being in \( L(M) \) does not necessarily mean that \( M \) eventually stops in a rejecting state. There is another possibility: \( M \) with input \( x \) does not eventually stop.

*Decide* implies *recognize*, but the converse is not necessarily true, as we shall see.

Decidability vs. Recognizability

A language is called *recognizable* if it is recognized by some machine. A language is called *decidable* if it is decided by some machine, i.e. recognized by a machine that always halts.

A language might be decided by one machine \( M \), but only recognized by another \( M' \). For example, we could modify the rejecting states of a deciding machine \( M \) to purposely going into an infinite loop, and call that machine \( M' \). Then \( M' \) only recognize the language that \( M \) decides.

Synonyms (for reading the literature)

*Recursive language* = decidable language  
*Recursively-enumerable language* = recognizable language  
*Recursive function* = computable function  
*Partial recursive function* = computable partial function
Partial Functions

A *partial function* is a binary relation $f$ having the property that

$$\forall x \forall y \forall z \ ((x, y) \in f \land (x, z) \in f) \rightarrow y = z.$$  

The $y$ such that $(x, y) \in f$ is typically noted $f(x)$.

What is *not required of a partial function*, but is required of a function, is that

$$\forall x \exists y \ (x, y) \in f$$

That is, for some $x$, $f(x)$ may be undefined in the case of a partial function. The use of this in computability is that a machine might compute a partial function but not a function because it *diverges* on some input $x$. In that case, we say "$f(x)$ is undefined", or $f(x) = \bot$. But $\bot$ is not a bona fide value, because we can’t test that is the result when the machine does not produce a result for input $x$. Other short-hands are: $f(x) \downarrow$ if $f(x)$ is defined, and $f(x) \uparrow$ if $f(x)$ is not defined. Sometimes people are less precise and use the word *function* when partial-function is intended. In this case, *total function* may be used to emphasize that this a true function and not just partial.

**Complement of a Language**

Suppose $L \subseteq \Sigma^*$ is a language. The complement of $L$ is denoted $L^c = \Sigma^* - L$.

**Corecognizability**

A language is called *corecognizable* iff its complement is recognizable.

(Thus a language is *recognizable* iff its complement is corecognizable.)

**Complementarity Theorem**

A language is decidable iff it is both recognizable and corecognizable.

One direction of this theorem is that, if a language is decidable, then its complement is also, by reversing accepting and non-accepting halting states.

The other direction is proved using *multiplexing*. Let $M$ be a machine recognizing $L$ and $N$ be a machine recognizing $L^c$. Given input $x$, run both $M$ and $N$ on $x$ simultaneously. One of them must accept $x$, because they are recognizers. They both won’t accept, as that would be contradictory. Whichever accepts, $M$ or $N$, indicates whether or not $x \in L(M)$.

![Proving the Complementarity Theorem](image-url)
**Core Unrecognizable Language**

Consider the language \{<M> | M accepts <M>\}. In other words, this is the language of descriptions of machines that accept their own description. This language is recognizable: Given <M>, construct <M, <M>> and run it on a universal machine.

Consider the complementary language, which we'll identify as K:

\[ K = \{<M> | M \text{ does not accept } <M>\} \]

i.e.

\[ K = \{<M> | <M> \notin L(M)\} \]

This language is *not* recognizable. We will argue this by contradiction.

Suppose K were recognizable. Let N be a machine recognizing K, i.e. \( K = L(N) \). Then we can ask the question of whether or not \(<N> \in K\).

If \(<N> \in K\), then by definition of K, \(<N> \notin L(N)\). But as \( K = L(N) \), we have \(<N> \notin K\), which is a contradiction.

Conversely, if \(<N> \notin K\) then \(<N> \notin L(N)\). But then \(<N> \in K\) by definition of K.

**Summary:**

\[ K = \{<M> | <M> \notin L(M)\} \text{ is not recognizable, and therefore not decidable.} \]

\[ K^c, \text{ the complement of K, is recognizable, but not decidable.} \]

**Reduction**

Reduction is a technique for showing languages to be undecidable or unrecognizable. As these are often negative results, the proofs demand a new way of looking at things. To say that a language L reduces to a language M, we mean that if M were recognizable, then L would also be recognizable. The means of showing this is to establish that we can determine whether or not a string is in L by transforming it to another string and determine whether the latter is in M.

An example of reduction involves the halting problem.

**Halting Problem**

The language

\[ \text{Halts}_{TM} = \{<M, x> | M \text{ halts when started on input } x\} \]

is not decidable.
In order to show this, we show that if $\text{Halts}_{TM}$ were decidable, then $K$ would be decidable (but it isn’t). Suppose that $<M>$ is a TM description that we want to know whether or not it is in $K$. We can create from $<M>$ by copying the pair $<M, <M>>$ and pass that to the machine for $\text{Halts}_{TM}$. It the answer comes back that $<M>$ halts on $<M>$, then we run a universal machine $U$ on $<M, <M>>$. If that determines $<M> \in L(<M>)$ then we know $<M> \notin K$, and if the universal machine determines $<M> \notin L(<M>)$, then we know $<M> \in K$. The following diagram illustrates the construction.

Reduction $K$ to $\text{Halts}_{TM}$: $<M> \in K$ iff $M$ does not accept $<M>$

Why the Halting Problem should really be called the Divergence Problem

If we examine the above argument, we can see that $\text{Halts}_{TM}$ is recognizable, because a universal machine whether $M$ halts on $x$ for arbitrary $M$ and $x$. It is the complement of $\text{Halts}_{cTM}$ that is not recognizable, i.e. whether $M$ diverges on $x$.

Mapping Reductions

Suppose $L \subseteq \Sigma^*$ and $M \subseteq \Delta^*$ are languages. We say that $L \leq_m M$, read $L$ is mapping-reducible to $M$, provided that there is a computable function $f: \Sigma^* \rightarrow \Delta^*$ such that

$$\forall x \in \Sigma^* \quad x \in L \iff f(x) \in M$$

In other words, an algorithm for deciding membership in $L$ can be obtained from an algorithm for deciding membership in $M$ along with an algorithm for $f$ that converts strings in $L$ to strings that are in $M$ and strings not in $L$ to strings not in $M$.

It is easy to see that $L \leq_m M$ implies:

- If $M$ is recognizable, then $L$ is recognizable.
- If $M$ is corecognizable, then $L$ is corecognizable.
- If $M$ is decidable, then $L$ is decidable.
However the following are not generally true:

If M is recognizable, then L is corecognizable.
If M is corecognizable, then L is recognizable.

So the sense of the mapping is important.

The use of \( L \leq_m M \) is typically in the *contrapositive* form:

If L is not recognizable, then M is not recognizable.
If L is not corecognizable, then M is not corecognizable.
If L is not decidable, then M is not decidable.

That is, we use to show a language M is not recognizable by identifying a known unrecognizable language L and showing \( L \leq_m M \). We say that we are reducing L to M (*not* reducing M to L). The key ingredient is that we have to find the computable mapping \( f: \Sigma^* \rightarrow \Delta^* \). This often takes the form of an algorithm for modifying a *machine description* that may or may not be in L into one that may or may not be in M.

**Accepts Notation**

Following Sipser, we define the acceptance language for Turing machines ("A" for *Accepts*)

\[
A_{TM} = \{<M, x> | M \text{ accepts } x\} = \{<M, x> | x \in L(M)\}
\]

**Question:** Why is \( A_{TM} \) recognizable?

**Mapping Reduction Example**

\( A_{TM} \) is not corecognizable (and therefore not decidable).

We can prove this by finding a reduction \( K^c \leq_m A_{TM} \), as \( K^c \) is not corecognizable (because K is not recognizable). That is, we need to find a computable function such that \( <M> \in K^c \) iff \( f(<M>) \in A_{TM} \). Notice that \( f(<M>) \) has to take the form \( <M', x> \) for appropriate \( M' \) and x, in order that \( <M', x> \in A_{TM} \). The question is what are \( M' \) and x?

For review, \( K^c = \{<M> | <M> \in L(M)\} \) and \( A_{TM} = \{<M', x> | x \in L(M')\} \).

To achieve the desired reduction, f must take as input the description of a machine M and produce a pair consisting of a machine \( M' \) with an input \( x \) to \( M' \) such that \( <M> \in L(M) \) iff \( <M', x> \in L(M') \).
In this example, we can take \( M' = M \) and \( x = <M> \), as the following are equivalent:

\[
<\!M\!> \in K^c \\
<\!M\!> \in L(M) \\
<\!M, <M>\!> \in A_{TM}
\]

In other words, the mapping function simply copies \( <M> \) to create a pair \( <M, <M>> \). The following diagram illustrates this construction. Note that in mapping reductions, yes from the inner box has to connect to yes of the outer box, and similarly for no. It cannot be reversed, as that would change the language being reduced into the complement of that language.

A key aspect of the above proof is that we map an arbitrary instance of \( K^c \) to a particular instance of \( A_{TM} \), not the other way around.

**Fixed-Tape Problem**

For each fixed input tape \( z \) there is a dedicated acceptance problem:

\[
A_{z_{TM}} = \{<\!M\!> | \ z \in L(M)\}
\]

This is obviously a special case of \( A_{TM} \) where the tape is always \( z \). So there is a trivial reduction \( A_{z_{TM}} \leq_m A_{TM} \). However, this does not prove that \( A_{z_{TM}} \) is undecidable. The inequality is the wrong direction for that.

\( A_{z_{TM}} \) is not corecognizable (and therefore not decidable).

We need to prove the opposite direction:

\[
A_{TM} \leq_m A_{z_{TM}}
\]
Let \(<M, x>\) be an instance of \(A_{TM}\). We need to map it to a corresponding instance of \(A_{z_{TM}}\). Here is the proposed mapping:

\[
\text{if}(<M, x>) = <M'>
\]

where \(M'\) first erases its input, then writes \(x\) on its tape and from then on behaves as \(M\) would on that tape. So \(M'\) will accept \(z\) iff \(M\) accepts \(x\). To see that \(f\) is computable, \(f\) must modify the description of \(M\) by adding additional rules that first erase \(M\)'s tape then writes \(x\) on the tape, then turns control over to \(M\). This gives the description of a new machine \(M'\).

![Reduction for the Fixed-Tape problem](image)

Note that constructions of this type do not necessarily imply that either \(M'\) or \(M\) ever get run as machines. It is up to the hypothetical program to analyze the program for \(M'\), but not necessarily run it.

**Blank-Tape Problem**

The blank-tape problem \(A_{\varepsilon_{TM}}\) is just the fixed-tape problem with \(z = \varepsilon\).

**Empty-Language Problem**

The empty language problem is whether the language recognized by a Turing machine is empty, i.e. the machine accepts no string. It sounds similar to the empty string problem, but should not be confused with it. However, the same construction can be applied.

Define

\[
\text{Empty}_{TM} = \{<M> \mid L(M) = \emptyset\}
\]

We show \(\text{Empty}_{TM}\) is not recognizable by showing \(\text{Empty}_{\varepsilon_{TM}}\) is not corecognizable.

Note that \(\text{Empty}_{\varepsilon_{TM}} = \{<M> \mid L(M) \neq \emptyset\}\). The same construction to the one in the fixed-tape problem to show that
\[ A_{TM} \leq_m \text{Empty}^c_{TM} \]

That is, \(<M, x> \in A_{TM}\) iff \(x \in L(M)\), and by the construction of \(<M'>\) from \(<M, x>\) in the fixed-tape problem, \(M'\) accepts some tape, i.e. \(L(M') = \emptyset\) iff \(<M, x> \in A_{TM}\).

**Question:** Is Empty\(_{TM}\) corecognizable? Equivalently, is Non-Empty\(_{TM}\) recognizable?

**Dovetailing for Non-Emptiness**

Suppose that \(<M>\) is the description of a TM and we want to determine whether \(M\) accepts some string. Let \(\{x_0, x_1, x_2, \ldots\}\) be an enumeration of all tapes over the alphabet of \(M\). We could try simulating \(M\) on \(x_0\), then on \(x_1\), then \(x_2\) etc. in turn to determine whether \(M\) accepts some string. If it does, then we know \(L(M)\) is non-empty. As we are only trying to recognize, rather than decide, those \(<M>\) that accept some string, this seems okay at first blush.

However, there is a problem with the above approach. Suppose \(M\) halts and does not accept \(x_0\), so we go on and try \(x_1\). It is possible that \(M\) diverges on \(x_1\), but still might accept \(x_2\). With the approach above, we’d never find that out.

The solution is to dovetail the computations of \(M\) on successive strings \(\{x_0, x_1, x_2, \ldots\}\). That is, try \(M\) for one step on \(x_0\). If that didn’t lead to acceptance, then try it for one step of \(x_1\), then go back and try one more step of \(x_0\). If none of those led to acceptance, try for one step of \(x_2\), then one more of \(x_1\) and \(x_0\), and so forth. If \(M\) accepts any string, then eventually one of the dovetailed simulations will accept, in which case we know that \(L(M)\) is non-empty. If \(M\) accepts no string, the dovetailed simulations will diverge. But halting is not required for recognizability if the input is not in the language, i.e. \(L(M)\) is empty, \(M \in \text{Non-Empty}_{TM}\).

**Effective Enumerability = Recognizability**

Call a language \(L \subseteq \Sigma^*\) effectively enumerable if there is a computable function \(f : N \rightarrow \Sigma^*\) such that \(L = \{f(i) \mid i \in N\}\), where \(N\) is the natural numbers. In other words, there is an algorithm that will compute the \(i^{th}\) element of \(L\) for any \(i\).

It is not hard to see that effective enumerability is equivalent to recognizability.

Suppose that \(L\) is effectively enumerable, to show that it is also recognizable: That there is a machine \(M\) such that with input \(x \in \Sigma^*\), if \(x \in L\) then \(M\) will halt and accept. (If not, we don’t care what happens; we are only looking for recognizability). \(M\) can operate as follows: It will contain the program for computing \(f\), and go about producing values \(f(0), f(1), f(2), \ldots\). If and when it reaches an \(i\) such that \(f(i) = x\), it halts and accepts \(x\). If it never reaches such an \(i\), then it will never accept \(x\), which is what we desire.
Suppose that \( L \) is recognizable, to show that it is also effectively enumerable, by a function \( f : \mathbb{N} \rightarrow \Sigma^* \). This is a little trickier. Suppose that \( N \) is a machine that recognizes \( L \). We want to use \( N \) to construct a machine that enumerates \( L \). Let \( \Sigma^* = \{x_0, x_1, x_2, \ldots\} \) enumerated in some standard order, such as by increasing length. We can run \( N \) on each element of \( \Sigma^* \) one after the other, maintaining a counter for the number of elements for which \( N \) accepts. When the counter reaches the value \( i \) that is the argument of \( f \), we declare that \( f(i) \) is the most recently accepted element of \( L \).

The problem here is that \( N \) could diverge on any given \( x_j \in \Sigma^* \), so we must dovetail those computations, rather than run them in sequence.

**“\( \text{All}_{\text{TM}} \)” Problem**

Another question that can be ask about a Turing machine is whether or not it accepts all strings. A closely related problem is whether it halts on *all* strings.

Once again, we can go back to the construction used for the fixed-tape problem and see that the language of descriptions of Turing machines that accept all strings is *not* recognizable. The machine \( M' \) constructed there accepts all strings iff it accepts any string. Thus

\[
A_{\text{TM}} \leq_{\text{M}} \text{All}_{\text{TM}}
\]

As \( A_{\text{TM}} \) is not corecognizable, \( \text{All}_{\text{TM}} \) is not corecognizable, and therefore not decidable. However, this does not establish that \( \text{All}_{\text{TM}} \) is recognizable, even though \( A_{\text{TM}} \) is. That would be using the reduction in the *wrong direction*.

Intuitively, \( \text{All}_{\text{TM}} \) is not recognizable, but we don’t have an obvious way to prove it yet.

**Is there a language that is neither recognizable nor corecognizable?**

We will show, by a new, tricky, technique, that \( \text{All}_{\text{TM}} \) is not recognizable. Since we already showed it is not corecognizable, \( \text{All}_{\text{TM}} \) is a language neither recognizable nor corecognizable.

**Step-Counting Technique**

We are going to reduce the unrecognizable language

\[
\text{Halts}_{\text{TM}}^c = \{<M, x> \mid M \text{ halts when started on input } x\}
\]

to \( \text{All}_{\text{TM}} \). That is, if \( \text{All}_{\text{TM}} \) is recognizable, then so is \( \text{Halts}_{\text{TM}}^c \). But we know that is false.

1. Let \( <M, x> \) be an arbitrary instance of \( \text{Halts}_{\text{TM}}^c \).
2. Create an instance \( <M'> \) of \( \text{All}_{\text{TM}} \) from \( <M, x> \) as follows:
a. $M'$, with an input $y$, runs $M$ on $x$ (not $y$) for just $|y|$ steps (or fewer, if $M$ halts earlier than $|y|$ steps.)

b. $M'$ accepts its input $y$ iff $M$ does not halt on $x$ in $|y|$ or fewer steps. Otherwise $M'$ rejects $y$.

3. Spelling out the above in detail:
   a. $M'$ accepts strings of length 0 iff $M$ does not halt on $x$ in 0 steps.
   b. $M'$ accepts strings of length 1 iff $M$ does not halt on $x$ in 1 step.
   c. $M'$ accepts strings of length 2 iff $M$ does not halt on $x$ in 2 steps.
   .
   .
   .

4. In other words that $M'$ accepts every input iff $M$ does not halt on $x$ in any number of steps, i.e. $M$ does not halt on $x$, period.

5. Thus a determination that $M'$ accepts every input is a determination that $M$ does not halt on $x$. In symbols:

   $$<M, x> \in \text{Halts}^c_{\text{TM}} \text{ iff } <M'> \in \text{All}_{\text{TM}}.$$ 

Furthermore, this is a mapping reduction, as $<M'>$ is effectively created from $<M, x>$.

**Rice's Theorem**

A *property* of a language is a condition that is true or false for the language, such as whether the language is finite. In order to ask whether a *language* has a property, we need a way to represent languages, and presenting a corresponding Turing machine that recognizes the language is a standard way. However, in order for this to be a property of a language and not the specific Turing machine recognizing it, we have to add the qualification of *functional* property.

To say that a property is *functional* means that the property does not depend on the specific machine used, but rather is shared by all machines, the languages of which have the property. The opposite of functional is *structural*: a property that depends on the structure of a specific machine.

To say that a property is *non-trivial* means that at least one machine has the property and at least one does not. In other words, the property is held by some, but not all, machines. A trivial property would be one that is always true or always false.

Examples of *non-trivial functional properties* represented by a machine $M$ are:

- $L(M)$ is finite
- $L(M)$ is regular
- $L(M)$ is decidable
Examples of non-functional properties held by a machine $M$ are:
- $M$ has fewer than 99 control states
- $M$ reverses direction fewer than 99 times on an empty input.

Now we can state Rice's Theorem:
Let $P$ be a non-trivial functional property of languages. Then $P$ is not decidable.

**Proof of Rice's Theorem:** Suppose $P$ is the property in question. For a reason to be seen, we need it to be the case that the empty set $\emptyset$ does not have property $P$. If instead $\emptyset$ does have $P$, then we replace $P$ with the complementary property and proceed. This is legitimate, because we are trying to show $P$ is undecidable, which will be true iff its complement is undecidable. For example, suppose we consider the property of being a regular language. The empty set is regular, so we must proceed with the property $P$ being non-regular instead.

Next, we make use of a machine that is known to have property $P$, call it $M_P$. This machine must exist because the property is assumed to be non-trivial. Because the property is also functional, it doesn’t matter which of many equivalent machines having property $P$ that we choose.

Now suppose we have machine that decides if an arbitrary $M$ has $P$. We are going to reduce $A_{\text{cTM}}$ to the language of the supposed $P$-decider. Given an instance $<M, x>$ of $A_{\text{cTM}}$, create a machine $M'$ such that $M'$ has $P$ iff $<M, x> \in A_{\text{cTM}}$, i.e. $M$ fails to accept $x$.

$M'$ on input $y$ behaves as follows: Run $M$ on $x$ (not $y$). *If that terminates,* behave as $M_P$ on the original input $y$. On the other hand, if $M$ does not terminate on $x$, $M'$ will not terminate either.

Now in what case does $M'$ have property $P$?

- If $M$ terminates on $x$: then $M'$ *has* property $P$, because it behaves as $M_P$ on the input to $M'$, and $M_P$ has $P$ by design.

- If $M$ does not terminate on $x$: then $M'$ *does not* have $P$, because the language of $M'$ is $\emptyset$, which does not have property $P$ by the consideration at the outset.

$M'$ is constructed effectively from $<M, x>$. If we have a decider for $P$, then we have a decider for $A_{\text{cTM}}$. But we already know this is impossible. Therefore $P$ is not a decidable property.