Chapter 19

Turing Machines, Recursively Enumerable Languages and Type 0 Grammars

19.1 Turing machines

We have seen that a pushdown automaton can carry out computations which are beyond the capability of a finite automaton, which is perhaps the simplest sort of machine able to accept an infinite set of strings. At the other end of the scale of computational power is the Turing machine (after the English mathematician, A. M. Turing, who devised them), which can carry out any set of operations which could reasonably be called a computation.

Like the previous classes of automata, a Turing machine can be visualized as having a control box, which at any point is in one of a finite number of states, an input tape marked off into squares with one symbol of the input string being inscribed on each square, and a reading head which scans one square of the input tape at a time. The Turing machine, however, can write on its input tape as well as read from it, and it can move its reading head either to the left or to the right. As before, a computation is assumed to begin in a distinguished initial state with the reading head over the leftmost symbol of the input string. We also assume that the tape extends infinitely to the left and right and that all tape squares not occupied by symbols of the input string are filled by a special "blank" symbol #.

The moves of a Turing machine (henceforth, TM) are directed by a finite set of quadruples of the form (q_i, a_j, q_k, X), where q_i and q_k are states, a_j is a symbol of the alphabet, and X is either an alphabet symbol or one of the special symbols L or R. Such a quadruple is interpreted in the following way: if the TM is in state q_i scanning a_j, then it enters state q_k (possibly identical to q_i) and if X is a symbol of the alphabet, it *replaces* a_j by that symbol. If X is L or R, then a_j is left unchanged and the reading head is moved one square to the left or right, respectively.

In the formulation we shall adopt, TM's are assumed to be deterministic; i e., for each state and each alphabet symbol there is at most one move allowed. We do not insist that there be a move for every state-symbol pair (this is similar to the formulation of deterministic pda's), and so if the TM reaches a point in its computation at which no instruction is applicable, it *halts*.

We note that a TM may in general read and write the blank symbol # and thus may extend its computation into portions of the tape beyond that originally occupied by the input string. Since it is not necessarily blocked by the #'s surrounding the input, one possibility open to a TM, but not to fa's or pda's, is that it might compute forever. This is an important property of TM's, as we shall see.

For example:

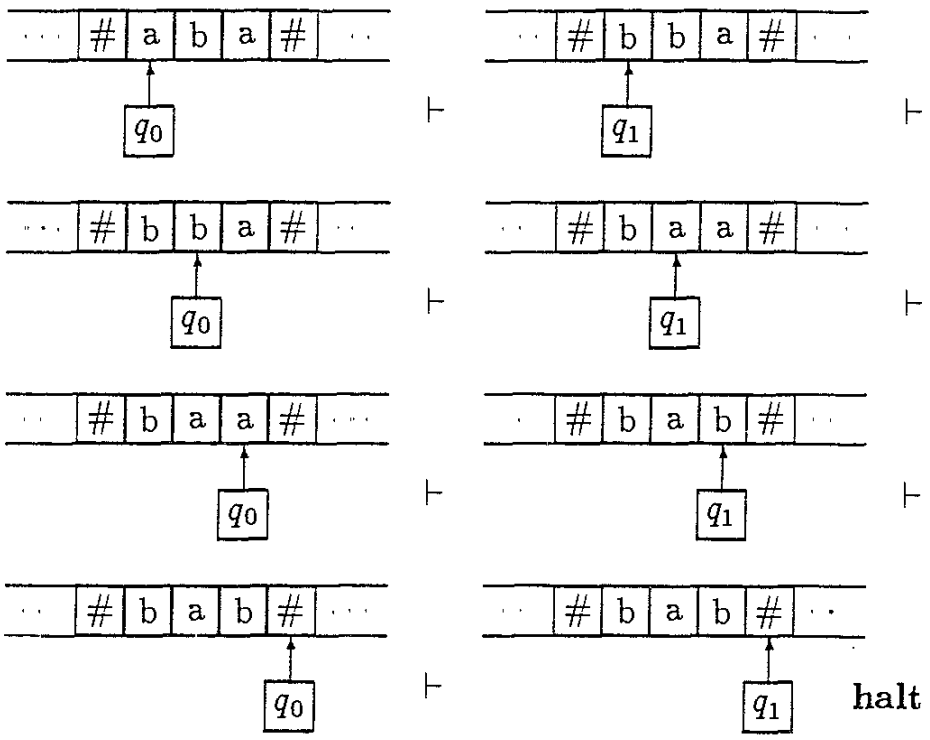
(19–1) The set of states K is $\{q_0, q_1\}$; the alphabet Σ is $\{a, b, \#\}$; the initial state is q_0; the set of instructions δ is written with an arrow between left and right halves for clarity:

$$\begin{aligned}
(q_0, a) &\rightarrow (q_1, b)\\
(q_0, b) &\rightarrow (q_1, a)\\
(q_0, \#) &\rightarrow (q_1, \#)\\
(q_1, a) &\rightarrow (q_0, R)\\
(q_1, b) &\rightarrow (q_0, R)
\end{aligned}$$

This machine scans an input string from left to right, changing a's to b's and b's to a's, until it encounters the first #. It then rewrites the # as #, changes state to q_1 and halts (since there is no instruction beginning $(q_1, \#)$). Since a # is sure to be found eventually, this TM has the property of halting on all inputs. Note, however, that if it had also contained the instruction $(q_1, \#) \rightarrow (q_1, R)$, then it would compute forever once it had reached the string of #'s to the right of the input. The same result could also be achieved by the instruction $(q_1, \#) \rightarrow (q_1, \#)$, except that instead

of scanning endlessly to the right, the TM would stay on one tape square forever reading and writing #.

(19–2) Example of computation:



We have not designated the states of a TM as final or non-final. It would be perfectly feasible to do so and to define acceptance of an input string in terms of halting in a final state. It will be slightly more convenient, however, to say that a string is accepted if, when it is given to the machine in the standard starting configuration, it causes the TM to halt after some finite number of moves; otherwise, it is rejected (i.e., the TM never halts). In (19-3) we give a machine which accepts all strings in $\{a, b\}^*$ which contain at least one a and which rejects, i.e., computes forever, when given anything else in $\{a, b\}^*$. (Note that we are here concerned only with strings in $\{a, b\}^*$ not $\{a, b, \#\}^*$. TM's are assumed always to be able to read and write the # symbol, but we will ordinarily confine ourselves to strings over alphabets which do not contain #. A TM which accepts some language in $\{a, b\}^*$ may give bizarre results when given a string not in this alphabet, but that doesn't matter. We are only concerned with its behavior when given inputs from

the relevant set)

(19–3) $M = \langle K, \Sigma, s, \delta \rangle$; $K = \{q_0, q_1\}$; $\Sigma = \{a, b, \#\}$; $s = q_0$;

$$\delta = \left\{ \begin{array}{ll} (q_0, a) & \rightarrow (q_1, R) \\ (q_0, b) & \rightarrow (q_0, R) \\ (q_0, \#) & \rightarrow (q_0, R) \\ (q_1, a) & \rightarrow (q_1, R) \\ (q_1, b) & \rightarrow (q_1, R) \end{array} \right\}$$

This machine scans left to right and stays in state q_0 so long as it sees b's. Once it encounters an a, it changes to state q_1 and continues rightward in this state until the first # and then halts It if meets the first # in state q_0, it scans right forever

19.1.1 Formal definitions

DEFINITION 19.1 *A* Turing machine M *is a quadruple* $\langle K, \Sigma, s, \delta \rangle$, *where* K *is a finite set of states,* Σ *is a finite set (the alphabet) containing* #, $s \in K$ *is the initial state, and* δ *is a (partial) function from* $K \times \Sigma$ *to* $K \times (\Sigma \cup \{L, R\})$. ■

A situation of a TM will be a quadruple of the form (x, q, a, y), where q is the current state, a is the symbol being scanned, and x and y are the strings to the left and right, respectively, of the reading head *up to the beginnings of the infinite strings of #'s*. This last provision is necessary to insure that a situation is uniquely specified. The TM in (19-2) is in situation (e, q_0, a, ba) at the beginning of the computation and in $(bab, q_1, \#, e)$ when it halts.

DEFINITION 19.2 *A situation of a TM* $M = \langle K, \Sigma, s, \delta \rangle$ *is any member* (x, q, a, y) *of* $\Sigma^* \times K \times \Sigma \times \Sigma^*$ *such that* x *does not begin with* # *and* y *does not end with* #. ■

We omit the formal definition of the *produces-in-one-step* relation, $\vdash_M$, on pairs of situation since it is rather complex when specified in full detail. Note that one must allow for cases such as the following: the TM is in situation $(abb, q, \#, e)$, as shown in Fig 19-1, and executes the instruction $(q, \#) \rightarrow (q', L)$. The resulting situation is (ab, q', b, e), as in Fig. 19-2, where the # originally being scanned has joined the infinite string of #'s to the right and has thus dropped out of the formal specification of the situation.

In a similar vein, if the instruction had been $(q, \#) \rightarrow (q', R)$, the resulting situation would be $(abb\#, q', \#, e)$ with a # taken from the string of #'s to the right and placed under the reading head.

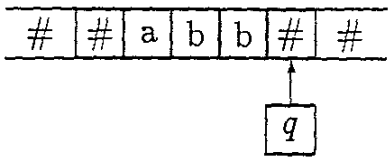


Figure 19–1.

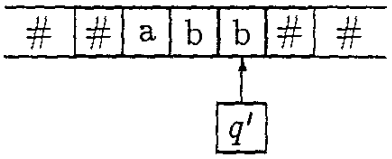


Figure 19–2.

Given the produces-in-one-step relation, we define the *produces* relation, $\vdash_M^*$ as its reflexive transitive closure.

DEFINITION 19.3 *Given a TM* $M = \langle K, \Sigma, s, \delta \rangle$ *and* Σ_1, *a subset of* Σ *which does not contain #, we say that* M accepts *a string* $x = a_1 a_2 \ldots a_n \in \Sigma_1^*$ *iff* $(e, s, a_1, a_2 \ldots a_n) \vdash_M^* (y, q, b, y')$, *where* y *and* y' *are strings in* Σ^*, $b \in \Sigma$, *and there is no instruction in* δ *beginning* (q, b) *(i.e., M has halted). (In case* $x = e$, *the initial situation is* $(e, s, \#, e)$.*)* ■

DEFINITION 19.4 *A TM* M accepts a language $L \in \Sigma_1^*$ *iff* M *accepts all strings in* L *and rejects (i.e., fails to halt on) all strings not in* L. ■

Note that we have defined acceptance so that it holds only of strings and languages defined over alphabets not containing #. This is primarily a technical convenience.

DEFINITION 19.5 *We say that a language is* Turing acceptable *iff there is some TM which accepts it.* ■

In virtue of Example (19-3) above, we can say that the language $L = \{x \in \{a,b\}^* \mid x \text{ contains at least one } a\}$ is Turing acceptable. In fact, all the regular languages and the deterministic and non-deterministic pda languages are Turing acceptable. (It should not be too difficult to imagine how one would go about constructing a TM to mimic the behavior of a fa or a pda.) Are there any languages, then, which are not Turing acceptable? There are, but it is not easy to exhibit one. We will return to this important question below.

We have seen that a given TM might not halt on certain inputs—indeed, we make use of this property in characterizing rejection of an input string. But this presents us with a problem. Suppose we have set a TM to computing on some input and it has not yet halted. Can we tell in general whether it is going to halt eventually or whether it is going to compute forever? This is the renowned Halting Problem for Turing machines, and we will show in a later section that there is, in general, *no* way to tell—at least if we mean by "a way to tell" some explicit procedure which can be computed mechanically by, say, a Turing machine. Another way to formulate the Halting Problem is this: Could we take any TM which accepts a language L and convert it into a machine which halts on all inputs (over the relevant alphabet) and signals its acceptance or rejection of the input by, say, the state it is in (or some special symbol printed on the tape, etc.)? Let us, in fact, make the following definition:

DEFINITION 19.6 *A TM $M = \langle K, \Sigma, s, \delta \rangle$ with some designated set $F \subseteq K$ of* final *states,* decides a language $L \subseteq \Sigma_1^*$, *(where $\Sigma_1 \subseteq \Sigma$ and does not contain #) iff for all $x \in \Sigma_1^*$, M halts in a final state if $x \in L$ and M halts in a non-final state if $x \notin L$.* ■

DEFINITION 19.7 *We say that a language is* Turing decidable *iff there is some TM which decides it, in the sense just defined.* ■

It is not difficult to convert the TM of Example (19-3) above into one which decides, rather than accepts, the language $L = \{x \in \{a,b\}^* \mid x \text{ contains at least one } a\}$—simply remove the instruction $(q_0, \#) \rightarrow (q_0, R)$ and designate q_1 as the only final state. The question is whether such a conversion can always be carried out. If so, then every Turing acceptable language is Turing

decidable, and, provided that the conversion can be carried out "mechanically", i.e., algorithmically, the Halting Problem is solvable. Since this turns out not to be so, we will have to conclude that there are Turing acceptable languages which are not Turing decidable. Again, most of the languages one ordinarily encounters as examples in formal language theory are Turing decidable, so one must look further afield to find one which is not.

Note that the implication in the opposite direction is easy to establish: every Turing decidable language is Turing acceptable. Given a TM which decides a language L, it is a simple matter to convert it into a TM which computes forever just in case the original machine would have halted in a non-final state. (Just add instructions which rewrite symbols as themselves while staying in the same state).

Turing machines are probably most often viewed not as language acceptors but as devices which compute functions. The initial input string is the argument for the function, and the expression on the tape when the machine halts (if it halts) is taken to be the value of the function at that argument. For example, the TM of (19-1) computes the function $f : \{a, b\}^* \rightarrow \{a, b\}^*$ such that when $f(u) = v$, v is like u with a's and b's interchanged. If the TM does not halt for a certain input, the function is not defined at that argument. Thus, TM's in general compute partial functions, but a TM which halts for all inputs in some set A computes a total function from A to its range. A TM which decides a language $L \subseteq \Sigma_1^*$, in the sense just defined, computes the characteristic function of L. A TM which accepts L computes a different function—one which is defined for all strings in L and undefined for all other strings in Σ_1^*.

By coding natural numbers as strings—say, in binary or in a unary encoding in which n a's represent the natural number n—we can let TM's serve a computers of functions from natural numbers to natural numbers. We can then ask whether functions such as $f(x) = x^2$ or $f(x) = x!$ are computable by Turing machine (they are, in fact, as are most of the functions ordinarily encountered). We can generalize this approach to functions of k arguments by letting the initial string given to the TM be, say, k blocks of a's of appropriate size separated by #'s. The TM which computes the addition function on natural numbers, for example, would start with $\ldots\#\#a^n\#a^m\#\#\ldots$ on its tape and end with $\ldots\#\#a^{n+m}\#\#\ldots$

Functions which are computable by Turing machine are called *partial recursive functions* (*partial*, because the TM may not halt for all arguments and thus may leave some values of the function undefined). The Turing

computable functions which happen to be total functions ought to be called "total recursive functions," but aren't. They're called simply *recursive functions*. A recursive function, thus, is a function which can be computed by a Turing machine which halts on all inputs in the domain of the function

A TM may also be regarded as a device for generating, rather than accepting or deciding, a set of strings. Let a TM be given as input some encoding of a natural number (say, n a's representing n) and let it compute until it halts, if it does. The contents of its tape between the infinite strings of #'s is some string over the alphabet of the TM (which may contain more than a's, of course), and this string is said to be generated by the TM. The set of all such strings generated, given all the natural numbers as inputs, is said to be the set *recursively enumerated* by the TM. A set is said to be *recursively enumerable* (abbreviated *r.e.*) if there is some TM which recursively enumerates it in the way just described

It turns out that a set of strings is r.e. just in case it is Turing acceptable. That is, if A is recursively enumerated by some TM T, there is a TM T' which accepts A, and conversely. (We will omit the proof of this result.)

Note also that a set which is recursively enumerated constitutes the range of a partial recursive function from the natural numbers to the set of all strings over the alphabet of A. In fact it is inessential here that the domain be the natural numbers—any denumerably infinite set would do. Thus, we can say that a set is recursively enumerable if it is the range of some partial recursive function. The following three statements, therefore, are equivalent:

(19–4) (i) A is accepted by some TM
(ii) A is the range of some partial recursive function.
(iii) A is recursively enumerated by some TM.

19.2 Equivalent formulations of Turing machines

There are many ways in which Turing machines can be defined which turn out to be equivalent in computational power. The input tape can be stipulated to be infinite in one direction only, or the machine can be endowed with any finite number of tapes which extend infinitely in one or both directions, or the machine may have one multiple-track tape with multiple reading heads. It may even be regarded as operating on an n-dimensional grid extending infinitely in all dimensions, so long as n is a finite number.

Even making a TM non-deterministic does not change its capabilities in any essential way. Suppose a non-deterministic TM had k distinct moves allowed from any given situation. Then from the initial situation there could be at most k possible situations after one step, at most k^2 after two steps, ..., at most k^n after n steps of the computation. A deterministic TM could keep track of all of these (at some expenditure of tape and time). If there is a halted computation by the non-deterministic machine, it occurs in some finite number of steps r. The deterministic machine will therefore discover this fact after having examined at most k^r possible situations—again, a finite number—and can then halt. A language accepted by some non-deterministic TM can therefore be accepted by a deterministic TM.

What seems to be essential to the formulation of a Turing machine, therefore, is that it has a finite number of states, a finite alphabet, a finite number of instructions, and an unbounded amount of computational space available to it.

19.3 Unrestricted grammars and Turing machines

An unrestricted (or Type 0) grammar $G = \langle V_T, V_N, S, R \rangle$ is one in which the only limitation on the form of the rules is that the left side contain at least one non-terminal symbol. Thus, letting upper case letters be non-terminals and lower case letters be terminals as usual, $aAbb \to ba$, $aAbB \to e$, and $A \to bCaB$ would all be allowed rules in such a grammar. Rules such as $ab \to ba$, $b \to BA$, or $e \to aA$ would be excluded. Note that because more than one symbol may be replaced in Type 0 rules, it is in general not possible to associate a phrase structure tree with a derivation in such a grammar.

Type 0 grammars can generate languages which are not context free. Here, for example, is an unrestricted grammar generating $\{x \in \{a,b,c\}^* \mid x$ contains equal numbers of a's, b's, and c's $\}$ which, as we have seen, is not a cfl.

(19–5) $G = \langle \{a,b,c\}, \{S,A,B,C\}, S, R \rangle$, where

$$R = \left\{ \begin{array}{llll} S \to SABC & AC \to CA & A \to a \\ S \to e & CA \to AC & B \to b \\ AB \to BA & BC \to CB & C \to c \\ BA \to AB & CB \to BC & \end{array} \right\}$$

This grammar works by producing strings of the form $(ABC)^n$ then permuting non-terminals freely by means of the rules $AB \to BA$, etc. Finally,

non-terminals are rewritten as the corresponding terminals. Here is a derivation of *cabbca*

(19–6) $S \Rightarrow SABC \Rightarrow SABCABC \Rightarrow ABCABC \Rightarrow ACBABC \Rightarrow CABABC \Rightarrow CABBAC \Rightarrow CABBCA \Rightarrow \ldots \Rightarrow cabbca$

And here is a Type 0 grammar generating the non-context free language $\{xx \mid x \in \{a,b\}^*\}$:

(19–7) $G = \langle \{a,b\}, \{S, S', A, B, \#\}, S, R \rangle$, where

$$R = \left\{ \begin{array}{lll} S \rightarrow \#S'\# & Aa \rightarrow aA & \#a \rightarrow a\# \\ S' \rightarrow aAS' & Ab \rightarrow bA & \#b \rightarrow b\# \\ S' \rightarrow bBS' & Ba \rightarrow aB & A\# \rightarrow \#a \\ S' \rightarrow e & Bb \rightarrow bB & B\# \rightarrow \#b \\ & & \#\# \rightarrow e \end{array} \right\}$$

Here is a derivation of *abaaba*:

(19–8) $S \Rightarrow \#S'\# \Rightarrow \#aAS'\# \Rightarrow \#aAbBS'\# \Rightarrow \#aAbBaAS' \Rightarrow \#aAbBaA\# \Rightarrow \#abABaA\# \Rightarrow \#abAaBA\# \Rightarrow \#abaABA\# \Rightarrow a\#baABA\# \Rightarrow a\#baAB\#a \Rightarrow \ldots \Rightarrow aba\#\#aba \Rightarrow abaaba$

This grammar works by generating sequences of aA's and bB's between #'s as endmarkers and then letting the non-terminals migrate to the right, where they can hop over the # and become terminals. The terminals in the left half similarly hop over the left end marker, and when the two #'s meet in the middle they are erased

The languages generated by the Type 0 grammars are exactly the languages accepted by Turing machines, i.e., the r.e. sets. We will not give detailed proofs of this equivalence here but will simply suggest how the proofs are constructed.

Given a Type 0 grammar G generating $L(G)$ a TM M accepting $L(G)$ can be constructed as follows. M is non-deterministic and has two tapes. Its input is given on the first tape where it is stored intact throughout the computation. The instructions of M essentially mimic the rules of G The initial symbol S is placed on the second tape, and M proceeds to rewrite as G would After the application of each rule, M compares the contents of the second tape with the input on tape 1. If they match, M halts, and thus accepts its input. If they do not match, M continues applying rules of G to

the string on tape 2, and if no rule is applicable, M cycles endlessly in some fashion. Clearly, if there is a derivation of the input string by G, there will be some computation by M which discovers this fact and thus M will halt and accept. If there is no such derivation, M computes forever, as required.

The simulation in the reverse direction—making a Type 0 grammar mimic a Turing machine—depends essentially on the fact that a situation of a TM can be regarded simply as a finite string of symbols and that to get from one situation to the next, some substring of these symbols is rewritten as some other string. For example, a TM instruction of the form $(q, a) \rightarrow (q', b)$ would correspond to the grammar rule $qa \rightarrow q'b$. Thus, situation (aab, q, a, bb) becomes (aab, q', b, bb) in one move by the machine, and, correspondingly, the string $aabqabb$ is rewritten as $aabq'bbb$ by the grammar. Left-moving and right- moving TM instructions require somewhat more complicated grammar rules to cover all possibilities. The details are tedious and not too instructive. Now, given a TM M which accepts language $L(M)$, we first convert it to a machine M' which behaves like M up to the point at which M would halt. M', however, replaces all non-blank symbols on its tape by #'s, then writes S, and halts in some designated state q_1. Thus, M' accepts $L(M)$ also but does so in such a way that it always halts in situation (e, q_1, S, e).

We now construct G so that it simulates the moves of M' in reverse. The initial symbol of G is S', and it first rewrites S' as $q_1 S$. Then, mimicking the moves of M' in reverse it can arrive at the string $q_0 x$, where q_0 is the initial state of M' and x was the input accepted. Now all G has to do is to erase q_0, thereby generating x. The only complication here is that we don't want to erase q_0 unless it is part of an initial situation of M'; i.e., q_0 might be entered at other points in the computation by M', and we don't want to erase it in these cases. This difficulty can be taken care of by adding new states to M' to insure that once it has left its initial state q_0 it never enters it again in the course of any computation. With this repair, the grammar G generates exactly the strings accepted by M' (and M). Thus, we can add a fourth equivalent statement to those given in (19-4) above:

(iv) A is generated by some unrestricted grammar

19.4 Church's Hypothesis

An *algorithm* is a fixed, deterministic procedure which can be applied mechanically to yield a result within a finite amount of time. For example,

there is an algorithm for finding the square root of any positive number to any desired number of decimal places. Algorithms are normally designed to apply to a *class* of problems, not to a single problem. The algorithm for finding square roots can be applied in the same way to *any* positive number—we do not have to hunt for a new procedure for every case.

Some classes of problems do not have algorithmic solutions. There is no algorithm, for example, for supplying proofs for theorems of geometry. To find a proof for a given theorem often requires some ingenuity, skill, and even luck, whereas algorithms, by definition, demand only simple clerical abilities.

The definition of algorithm given above is not mathematically precise, relying as it does on such intuitive notions as "mechanical." In the 1930's, a number of attempts were made to find a precise, formal characterization of the notion of algorithm as applied to mathematical problems. The Turing machine was the result of one such attempt. It is clear that a Turing machine satisfies our intuitive notion of what an algorithm should be; a TM which computes a function, for example, determines the value at any agrument in a fixed, deterministic, mechanical way and in a finite amount of time. The question then arises whether all things which we would intuitively call algorithms can be formulated as Turing machines. The conjecture that this is indeed the case has been given the name *Church's Thesis* or *Church's Hypothesis* (after the logician Alonzo Church). It is not a theorem, since it relates a mathematical construct—the Turing machine—to an intuitive, imprecise notion—an algorithm. It is nonetheless widely believed to be correct. The evidence in its favor arises basically from the fact that all independent attempts to characterize the notion of algorithm by mathematicians such as Kleene, Post, Markov, Church, and others turned out to be equivalent to the Turing machine. Rogers (1967) calls this the Basic Result in Recursive Function Theory:

> The classes of partial functions (and hence total functions) obtained by the characterization of Turing, Kleene, Church, Post, Markov, and certain others, are identical, i e , are just one class.

Further support for Church's Hypothesis comes from the fact that modifications and enrichments to the definition of a Turing machine which keep intact our view of it as a mechanical computing device with a finite number of states and instructions but with a potentially unlimited amount of space for computation, always produce an equivalent device.

These results suggest that the characterizations are not purely arbitrary but do in fact define a natural concept, i.e., sets which are recognizable, or functions which are computable, by algorithm If we accept Church's Hypothesis, then, we may add a fifth equivalent statement to the list in (19-4) above.

(v) There is an algorithm for recognizing strings in A.

19.5 Recursive versus recursively enumerable sets

A Turing machine which accepts a set A halts eventually whenever given a member of A as input but fails to halt when given a non-member of A. If we accept Church's Hypothesis, this means that an algorithm may work in such a way that it yields an answer in a finite amount of time for all members of a particular class but may yield no result for things not in the class. It may happen, however, that there exist algorithms for recognizing not only all members of A but also one for recognizing all members of $\Sigma^* - A$, i e., the complement of A. When this is so, A is called a *recursive set*. To state the definition formally in terms of Turing machines, a set A is recursive iff both A and A' are recursively enumerable (= Turing acceptable). It now follows that the recursive sets are just the Turing decidable languages defined above. Recall that a language is Turing decidable if there is some TM which halts on all inputs over the relevant alphabet and signals whether or not the input was in the language. As we have seen, it is a simple matter to convert a TM which decides L into one which accepts L (or into one which accepts L') by causing it to compute forever on negative outcomes. Conversely, if we were given two TM's, one accepting L and one accepting L', we could construct a TM for deciding L by having it simply alternate the instructions of the two machines on two copies of the input One of these will eventually halt, since by assumption both L and L' are Turing acceptable, and then our composite machine can signal whether the input was in L or L'. Thus it decides L.

Analogous remarks could be made about computation of functions rather than recognition of sets. The functions which are algorithmically computable are, by Church's Hypothesis, just the parital recursive functions. If the function is properly partial and not total, the algorithm will yield no value at arguments for which the function is not defined If the function is total, however, the algorithm yields a value at each argument As we have seen, a TM which decides a language in effect computes the characteristic function of

that language. The recursive sets, then, are just those which have recursive characteristic functions. The recursively enumerable sets have characteristic functions which are partial recursive functions. We may summarize our statements about recursive sets as follows:

(19–9) The following statements are equivalent:

(i) A is a recursive set

(ii) A is Turing decidable

(iii) Both A and A' are recursively enumerable

(iv) A has a characteristic function which is (total) recursive.

We do not yet officially know, of course, whether there are actually any r.e. sets which are not recursive. We turn our attention to this matter in the next section.

19.6 The universal Turing machine

Since a Turing machine is defined as a finite set of quadruples together with a designated initial state (the set of states and the alphabet are implicit in the quadruples), it is possible to enumerate all possible Turing machines To be somewhat more explicit, we might code the states and alphabet symbols as sequences of 1's and separate them by 0's. A complete coding for a Turing machine might, then, look something like this:

We might also agree on a fixed order of listing the quadruples so that each TM has a unique representation in this coding scheme. We could now enumerate TM's by listing them with machines with the smallest number of quadruples first in increasing order according to their encodings interpreted as a binary number. Thus we have a one-to-one correspondence between TM's and the natural numbers. (Incidentally, this fact shows us that there must be languages which are not Turing acceptable simply by considering the cardinalities of the sets involved. Given a finite alphabet A, there are $\aleph_0$ strings in A^*. There are $2^{\aleph_0}$ subsets of A^*, i.e., languages over the alphabet A. By Cantor's Theorem, $2^{\aleph_0} > \aleph_0$ and since there are only $\aleph_0$ Turing machines, there is an uncountable infinity of languages over any given alphabet which are not Turing acceptable.)

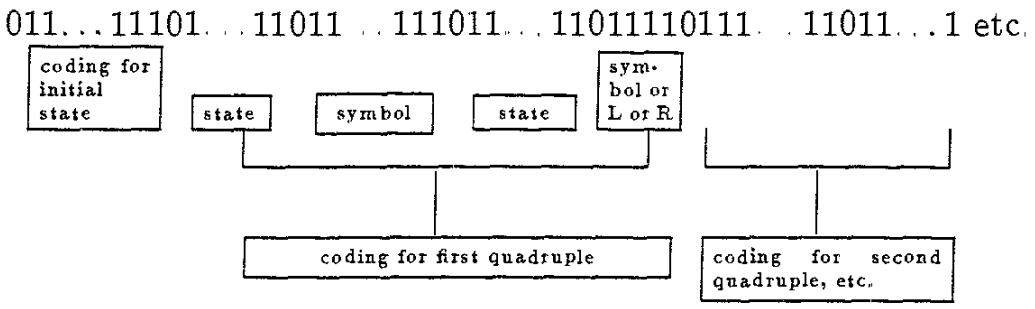


Figure 19–3.

Let us also assume that input strings are encoded into 1's and 0's in some fixed fashion. We may also assume without loss of generality that whatever output a TM leaves on its tape can also be encoded into 1's and 0's. Let us denote by $E(M)$ an encoding of a Turing machine M and by $E(x)$ the encoding of an input string x. It now turns out that there is a TM U, the universal Turing machine, which can take as input $E(M)E(x)$ and mimic the behavior of M on x. That is, if M halts on x, U halts given $E(M)E(x)$ and leaves on its tape an encoding of whatever output M would have left when it halted; if M does not halt on x, then U does not halt on $E(M)E(x)$

We will not attempt to give the construction of U here, but it can be thought of as a three-tape machine which keeps on its tapes (1) the encoded instructions of M, (2) an encoded version of the non-blank portion of the tape M would have at each point in its computation, and (3) an encoded representation of the current state. U consults tape 3 for the current state, examines tape 2 to see which symbol is under the reading head (of M) and then consults tape 1 to find the instruction beginning with that state and symbol. If none is found, M would have halted, and U halts. If one is found, U makes the appropriate changes to tape 2, changes the state on tape 3 and repeats the cycle.

To dispel any possible air of mystery surrounding the universal Turing machine, let us point out that simulating the moves of any given TM on any given input tape falls under the class of procedures which can be carried out in purely mechanical fashion. (Think what would be involved if you were asked to carry out this task yourself.) It is therefore executable by a

Turing machine. The universal character of the machine arises from the fact that all Turing machines and input tapes are given an encoding over a fixed alphabet (here, 1's and 0's, although any convenient alphabet would do). Thus U needs to be programmed only to find instructions and carry them out on tapes all coded in the same way.

Another way of formulating what we have just said is that the language $\{E(M)E(x) \mid M \text{ accepts } x\}$ is Turing acceptable. This is a language over the alphabet $\{1,0\}$, and it is accepted by the Turing machine U (actually, a slight variant of U which first checks to see if the string which it has received as input is in fact of the right form to be a TM encoding followed by a string encoding; this again is an easily arranged mechanical procedure)

19.7 The Halting Problem for Turing machines

We are now ready to address the problem mentioned above, namely, the problem of deciding for an arbitrarily given TM and an arbitrarily given input string whether the TM will ever halt on that input. Given our method of encoding TM's and input strings, it is easy to state the halting problem in terms of Turing acceptable and Turing decidable languages. We have just seen that the language

(19–10) $L = \{E(M)E(x) \mid M \text{ accepts } x\}$

is a Turing acceptable language. Thus, in order to determine whether M halts given x, simply give the encoding of M and the encoding of x to (modified) U. If M halts on x, U will also But now if M does not halt on x, U doesn't halt either. We want to know if there is some way to tell that M will *not* halt on x when that is the case. In other words, is L *Turing decidable*? Is there some TM which will halt and say yes if M halts on x, and will halt and say no when M does not halt on x?

We will show that this cannot be the case. Assume L is Turing decidable by some TM M_L. Since L is decidable for all $E(M)$ and $E(x)$, it will be decidable in the special case in which x happens to be $E(M)$ itself. It may seem strange to give a Turing machine its own encoding as an input tape, but since this encoding is just a long string of 1's and 0's, there is nothing in principle to prevent us from doing so. That is, the following language L_1 is Turing decidable if L is:

(19–11) $L_1 = \{E(M) \mid M \text{ accepts } E(M)\}$

It would in fact be decided by a TM M_1 which first encodes its input and copies it directly to the right of the original and then behaves like M_L

But now, since L_1 is decidable, it follows that its complement is decidable and hence, Turing acceptable; i.e., there is a TM, call it M^*, which accepts L_1'.

(19–12) $L_1' = \{x \in \{0,1\}^* \mid x \text{ is not a TM encoding, or else } x \text{ is the encoding of a TM } T \text{ and } T \text{ does not accept } E(T) \text{ (i.e., } x)\}$

We now ask whether the encoding of M^* itself is in L_1'; that is, is M^* a machine which does not accept its own encoding as input?

First case: $E(M^*) \in L_1'$. Then $E(M^*)$ is one of the strings accepted by M^*, by the assumption that M^* accepts L_1'. So M^* accepts $E(M^*)$. But because $E(M^*) \in L_1'$, it is the encoding of a Turing machine which does *not* accept its own encoding, i e., M^* does not accept $E(M^*)$. Contradiction

Second case: $E(M^*) \notin L_1'$. Then $E(M^*)$ is *not* a string accepted by M^*, by the assumption that M^* accepts L_1'. So M^* does not accept $E(M^*)$. Therefore, M^* is a TM which does not accept its own encoding; therefore $E(M^*)$ is a member of L_1'. Contradiction.

Since $E(M^*)$ must either be in L_1' or not in L_1', and either assumption leads to a contradiction, we conclude that there is no such machine as M^*; thus, L_1' is not Turing acceptable (our first example of a set of this sort). But the Turing acceptability of L_1' was implied by its Turing decidability, which in turn was implied by the Turing decidability of L_1 Therefore, we conclude that L_1 cannot be Turing decidable after all. Finally, L_1's Turing decidability was implied by the assumed Turing decidability of L. Thus, we conclude that L is not Turing decidable, and the Halting Problem for Turing machines is not decidable by Turing machine; hence, given Church's Hypothesis, not decidable by algorithm

Note that in the process of proving the undecidability of the Halting Problem, we have exhibited a set which is not Turing acceptable (namely, L_1') and sets which are Turing acceptable but not Turing decidable (namely, L and L_1). We can thus state the following:

THEOREM 19.1 *There are sets which are not recursively enumerable.* ■

THEOREM 19.2 *There are sets which are recursively enumerable but not recursive.* ■

As we have seen, these are not ordinary garden variety sets. The latter are exemplified by the set of all encodings of TM's which accept their own encodings as input; the former by the set of all Turing machines (in encoded form) which do not accept their own encodings as input. However, having established a foothold in this territory we can use these sets to discover others of their class. We may also use the undecidability of the TM Halting Problem to prove that other problems are undecidable as well. For example, we will show the following problem for Turing machines to be undecidable:

Problem: For an arbitrarily given TM M, does M halt given e, the empty string, as input?

We first express the problem as a language:

(19–13) $L_2 = \{E(M) \mid M \text{ accepts } e\}$

and ask whether there is a TM M_2 which *decides* this language. We show that there is not, and the proof technique is to show that if such a machine existed then it could be modified to produce a machine which decides the Halting Problem. Since the latter cannot exist, neither can M_2.

Suppose we have M_2, which by hypothesis decides L_2. We show how to use M_2 to construct a machine M, which decides L, where L is the language of the Halting Problem:

(19–14) $L = \{E(M)E(x) \mid M \text{ accepts } x\}$

First of all, for any given TM M and any given input string x, one can modify M so that if it is started on the empty tape it will first write x on it and then proceed as M would have, given x. Call this modified machine M_x. M_x first checks to see if its input is the empty tape. (If not, it runs forever in some fashion.) If so, it writes x (a finite string, so this is done by some finite set of instructions added to M) and then positions its reading head for the start of a computation and executes the moves of M thereafter.

Now if we assume that we have a machine M_2 which decides L_2, then it will work, in particular, if it is given the encoding of any machine M_x

constructed in the way just described. (M_2 works for *any* TM encoding; it will work if that encoding happens to be the encoding of M_x.) But notice that M_x accepts e just in case M accepts x. That is, M_x halts, given the empty string as input, iff M halts given x. Therefore a machine which decides whether M_x halts given e could be used in effect to decide whether M halts given x. This has been shown to be impossible so there is no such machine as M_2, and language L_2 is therefore not Turing decidable.

A whole host of problems concerning Turing machines turn out to be undecidable: whether an arbitrarily given TM ever enters a particular state, whether it halts on any inputs at all, whether it halts on every input, whether it ever writes a particular symbol on its tape, etc. These and other undecidability results can, in turn, be used to establish undecidability results in other areas. For example, the undecidability of the TM Halting Problem can be used to establish the undecidability of another problem called the Post Correspondence Problem. This can then be used in showing that certain problems concerning context free grammars and languages are undecidable. For example, it is undecidable, given two arbitrary cfg's G_1 and G_2 whether $L(G_1) = L(G_2)$, whether $L(G_1) \cap L(G_2) = \emptyset$, whether $L(G_1) \subseteq L(G_2)$, whether $L(G_1)$ is inherently ambiguous, etc.

One should note carefully that none of these undecidability results imply that for a particular TM, a particular cfg, etc. there is no way to determine whether it halts given the empty string, whether it is inherently ambiguous, etc. We have seen examples of TM's, e.g., in (19-1), which can be shown, quite easily in fact, to halt given the empty string. What the undecidability result says is that there is no single, generally applicable algorithm which is guaranteed to work for every TM (or every arbitrarily given pair of cfg's, etc.)

It is also worth noting that in view of the correspondence shown above between TM's and unrestricted grammars, the undecidability results for TM's can be carried over immediately to grammars. Thus, there is no algorithm for determining for an arbitrarily given Type 0 grammar G whether G generates any strings, whether G generates the empty string, whether G generates all strings in Σ^*, etc.

Exercises

1. Construct a Turing machine that accepts any tape written on the vocabulary $\{0, 1\}$ and converts every contiguous string of two or more 1's

to 0's. Everything else is left unchanged. For example, the input tape $\cdots\#01011011101\#\cdots$ should end up as $\cdots\#01000000001\#\cdots$.

2. Construct a Turing machine with three states $\{q_0, q_1, q_2\}$, initial state q_0, that begins with an input tape consisting entirely of blanks and halts with exactly three contiguous 1's on the tape.

3. Consider the following Turing machine: $M = \langle\{q_0, q_1\}, \{a, b, \#\}, q_0, \delta\rangle$ where

$$\delta = \left\{ \begin{array}{l} (q_0, a) \rightarrow (q_0, R) \\ (q_0, \#) \rightarrow (q_1, a) \\ (q_1, a) \rightarrow (q_1, L) \\ (q_1, \#) \rightarrow (q_0, a) \end{array} \right\}$$

(a) Write the first twelve situations of the machine M if it starts in the situation $(e, q_0, a, \#\#a)$.

(b) Describe verbally what machine M will continue to do after this much has been done.

(c) Will it ever halt?

(d) Will it use only a finite amount of tape?

(e) Are there any squares of the tape that it will scan only a finite number of times?

(f) What will machine M do if started in another situation?

4. **(a)** Make up a simple Turing machine which never halts no matter what the initial tape sequence is. Give both the quadruples and a verbal description of its behavior. Let the machine be allowed to start scanning at any square but always start in state q_0.

(b) Similarly, make up a Turing machine which always halts eventually.

5. Tell whether the following functions are total or only partial A function is considered to be undefined if it would yield a value outside the set on which it is specified

(a) Addition on the set of all even integers.

(b) Addition on the set of all prime numbers.

(c) Set union on the set $\{\{0\}, \{1\}, \{2\}, \{0, 1\}\}$

6. Describe informally an algorithm for converting an integer in binary notation to decimal notation.

7. Write a Type 0 grammar generating the language $\{a^{2^n} \mid n \geq 0\}$.

8. Show that the following problem for Turing machines is undecidable: For an arbitrarily given TM M, does M accept at least one string? (Hint: Show that if a TM existed which decided the language $\{E(M) \mid M$ accepts at least one string$\}$, it could be modified to produce a machine deciding L_2 in (19-13).)