For the students of CIS 682, Fall 2015. PLEASE DO NOT DISTRIBUTE FURTHER

Lecture Notes on Computability

©Val Tannen 1995, slightly revised in 2015

1 Models of Computation

From the working computer scientist's point of vue, Computability Theory is concerned with the mathematical limitations on what can be programmed. Any treatment of the subject begins with a mathematical definition of the "computing agents". Imprecisely, a computing agent is a program. A survey of different approaches to Computability Theory will show definitions of computing agents that vary from those resembling (some) Pascal programs to those resembling machine language programs, to those resembling nothing technologically existent, such as Turing Machines.
For various reasons, most texts on the subject begin with a description of Turing Machines, and these notes will follow this tradition. This is useful especially if one also studies Complexity Theory, where the intimate functioning of Turing machines plays a more important role. I believe however, that the intimate workings of the computing agents play a significant role only in the most boring parts of Computability Theory, parts which amount to lots of laborious programming in usually very weak programming languages. Relying on programming experience, we can skip most of these boring parts, with the knowledge that, if needed, we can supply the missing and gory details. After some familiarization with the subject, we will find that we are quite comfortable without ever seeing those details.

1.1 Turing Machines

The treatment of Turing Machines follows "Introduction to Automata Theory, Languages, and Computation", by J. E. Hopcroft and J. D. Ullman, Addison-Wesley, 1979. A textbook that uses a version of While Programs, albeit computing with natural numbers rather than strings, is "A Programming Approach to Computability", by A. J. Kfoury, R. N. Moll, and M. A. Arbib, Springer-Verlag, 1982.
In the Turing Machine model of computation, a computing agent, i.e., a Turing machine, consists of

- a tape divided in cells and infinite in one direction (say, to the right), such that each cell can hold a symbol (character) from a finite alphabet associated with the machine
- a "finite control", that is, a finite set of states and a "next move" function

- a tape head that scans one cell at a time.

In one move the machine, depending on the state of the finite control and the symbol in the tape cell currently scanned by the tape head will

- change state
- writes a symbol in the cell that's currently scanned
- move the tape head left or right one cell.

Formally, a Turing machine M is a tuple

$$M = (\Gamma, \Sigma, \#, Q, \delta, q_0, F)$$

where Γ is a finite *tape alphabet*, $\Sigma \subseteq \Gamma$ is an *input alphabet*, $\# \in \Gamma$ is the *blank* ($\# \notin \Sigma$), Q is a finite set of *states*, $\delta : \Gamma \times Q \longrightarrow Q \times \Gamma \times \{L, R\}$ is a *partial* function that gives the *next move*, $q_0 \in Q$ is the *initial* state, and $F \subseteq Q$ is a set of *final* states.
We describe next the conventions used to make Turing machines *accept* strings (words) over the alphabet Σ. Initially, the tape contains a word (the input) on the tape, in the leftmost cells, while the rest of the tape contains blanks. This way, only a finite portion of the tape is f interest, and this stays true after each move, hence, we can describe the state of the machine's store, which is the tape, using only a finite amount of information. The finite control starts in the initial state. A sequence of moves follows. We have described above of what consists a move. The machine may halt in two situations: first when δ is undefined (recall that δ is partial) on (γ, q) where γ is the symbol currently read by the tape head and q is the current state; and second, when it tries to move the head left from the first cell. The second situation can be avoided by programming conventions such as having a special symbol to the left of the input in the first cell. We say that the input word is accepted if the finite control reaches a final state. Programming conventions may insure that the machine also halts in that case. Other programming conventions can insure that the machine never halts in a non-final state (how do we do this?).
Here is a formal definition of the predicate "machine M accepts word w", where $M = (\Gamma, \Sigma, \#, Q, \delta, q_0, F)$ and $w \in \Sigma^*$. Without loss of generality, we can assume that Q and Γ are disjoint. An *instantaneous description* (ID) of M is a string uqv where $q \in Q$ is the current state of the finite control, and $uv \in \Gamma^*$ is the content of the tape up to the rightmost nonblank symbol or the symbol in the cell to the left of the head, whichever is rightmost, such that the tape head is scanning the leftmost symbol of u, or, if $v = \text{nil}$, the head is scanning a blank. It is straightforward to define a *one-move relation* $\vdash$ between ID's. For example, if $\delta(\gamma, q) = (\gamma', q', L)$ then for any $u', v' \in \Gamma^*$

$$u'\gamma''q\gamma v' \vdash u'q'\gamma''\gamma' v'$$

except if $v' = \text{nil}$ and $\gamma' = \gamma'' = B$ in which case

$$u'\gamma''q\gamma v' \vdash u'q'$$

To insure that the tape head will not "fall off" the left end of the tape, if $\delta(\gamma, q) = (\gamma', q', L)$ then the ID's of the form $q\gamma v'$ are not related to any other ID by the one-move relation. We define the *several-moves relation* to be the transitive-reflexive closure of $\vdash$, notation $\vdash^*$. Then, we say that the word $w \in \Sigma^*$ is accepted by M whenever there exists a final state $f \in F$ and some (uninteresting) content of the tape $u, v \in \Gamma^*$ such that

$$q_0 w \vdash^* ufv$$

We will shortly claim that one can compute with a Turing machine anything that can be computed with, say, a Pascal program. This result can be proved rigorously but we feel that such proof are not terribly informative. We will only indicate a couple of Turing machine "programming tricks" which go some way toward making available the usual tools of Pascal-like programming. We rely on your programming experience for you to be convinced that the remaining details can be tediously but straightforwardly supplied.

The first trick is that we can "store" symbols in the finite control, which allows us to transport them between two tape cells. This is done by defining the set of states as a cartesian product $Q = Q' \times \Gamma$ so each state has the second component in which a symbol can be "stored" while the first component takes care of the control flow. Another trick allows us to "check off" the symbols in certain cells. This useful if the head must go back and forth between cells covering a distance that depends on the various inputs. It's done with a tape alphabet of the form $\Gamma = \Sigma \cup \Gamma' \times \{\surd, -\}$ where Γ' is (conceptually) the tape alphabet that is used beyond the checking off issue (it will be useful to have $\Sigma \subseteq \Gamma'$), $\surd$ means "checked"and $-$ means "unchecked". A slight generalization of this idea allows us to have the equivalent of auxiliary storage (beyond that for the input, that is). This is the same as having a machine with more then one tape so we might as well define such a model of computation and show that it doesn't bring more power (but it brings more convenience in definitions and proofs).

A *multitape Turing machine* has one finite control, but several tapes, each with its read-write head, moving independently. Each tape can have its own alphabet. Note that the number is fixed for a given machine, there is no new tape creation during the functioning of the machine (as opposed to programs in which new variables can be declared dynamically).

In one move the machine, depending on the state of the finite control and the (tuple of) symbols in the tape cells currently scanned by the tape heads will

- change state
- write a symbol in each of the cells that are currently scanned by the tape heads (some of these symbols can be the same as the existing ones which amounts to no activity by the corresponding heads
- move each of the tape heads left or right one cell, or just leave it where it was.

The input alphabet must be a subset of the alphabet of the first tape so we can adopt the convention that initially the input is on the first tape. Of course, the first tape is otherwise blank, and so

are the other tapes. It would be straightforward to generalize the formalism of the definition of the one-tape Turing machine but we feel that this would not improve the understanding of the definition, nor of the main theorem presented next.

Theorem 1.1
For any multitape Turing machine MM there exists a one-tape Turing machine M which simulates the behavior of MM. In particular, M accepts exactly the same words as MM.

Proof. (Sketch) To simplify the notation let's say that MM has three tapes, and let $\Gamma_1, \Gamma_2, \Gamma_3$ be their alphabets. Let Σ be the input alphabet. Then, the tape alphabet of the equivalent one-tape machine is

$$\Gamma = \Sigma \cup (\Gamma_1 \times \{H_1, N\} \times \Gamma_2 \times \{H_2, N\} \times \Gamma_3 \times \{H_3, N\})$$

Think of the tape of M as having 6 "tracks", three of them used to simulate the three tapes of MM and the other three used to mark the position of the three heads (H_k means "head of tape k is here" while N means "no head is here).
To simulate a single move of MM, the machine M will sweep its tape from left to right up to the rightmost H_k, figuring out what simulated symbols are the simulated heads of MM reading. This is possible because the finite control of M is built with the knowledge of the fixed number of tapes (in the illustration, 3) of MM. The state of M will store the state of MM as one of its components so M can now figure out what MM will write on its tapes and how it moves its heads (in some sense, the next-move function of MM is built into that of M) and record it during a sweep back toward left. Finally M will change its state to reflect the new state of MM. If the new state is final, then M's state will be final too. **End of Proof.**
The language of words accepted by a Turing machine is one of several ways to describe its computing power. We can also use Turing machines to compute partial functions, an activity that is closer in generality to usual programming:

Definition 1.2 *Fix two alphabets, an "input" alphabet Σ_i, and an "output" alphabet Σ_o. A partial function $f : \Sigma_i^* \dot{\to} \Sigma_o^*$ is said to be* **Turing-computable** *if there exists a Turing machine M such that for any $u \in \Sigma_i^*$, M halts on input u iff f is defined on u, and when M halts on input u, it outputs $v = f(u)$. In such a situation, we say that M* computes *f.*

Note that the function in the definition above is a function of one argument. In the theory of recursive functions, one usually considers functions of several arguments which are natural numbers. In our generalization from natural numbers to strings, the distinction between one and several arguments is unimportant, because we can trivially encode a tuple of strings as one string, simply by concatenating them with some special separator symbols in between. We should note however that separating the arguments and even the result is often very convenient, and the multitape Turing machine offers an easy way to do this: one could say that a function of n arguments is computed by a multitape Turing machine with n+1 tapes if in the beginning the arguments are on the first n tapes and at the end of teh computation teh result is on the last tape.

1.2 While Programs

According to an intuitive ordering based on "expressive power", while-programs are somewhere between Turing machines and, say, (idealized) full Pascal programs. We will state, without proof, that Turing machines have the same "computing power" as Pascal programs. It's easier to believe this if you believe, first, that the Turing machine programming tricks that we saw previously would alow us to simulate while-programs, and second, that while-programs can simulate (have them as "syntactic sugar") all the other features that Pascal programs have. We will give some evidence for the second part below.

Here is the syntax of while-programs, in BNF:

< identifier >::= ... (your favorite names convention)

< char >::= 'a' | 'b' *etc.*

< expression >::=< identifier >|< char >| NIL | TAIL(< expression >) | CONS(< char > , < expression >)

< test >::=< expression > = < expression >| NOT(< test >) | OR(< test > , < test >)

< statement >::=< identifier > := < expression >| SKIP | WHILE(< test > DO < statement − sequence > OD

< statement − sequence >::=< statement >|< statement > ; < statement − sequence >

< program >::=< statement − sequence >

Note that variables can contain only values that are strings of characters (we use list-like notation for the empty string, for taking the tail or for adding a character at the beginning of a string—cons-ing). SKIP does nothing. The tail of the empty string equals, by convention, the empty string.

Other familiar programming constructs can be added to the language as syntactic sugar. In other words, we can simulate more complicated constructs with while-programs. Strings, such as 'abc' are equivalent to

```
CONS( 'a' , CONS( 'b' , CONS( 'c' , NIL )))
```

Using De Morgan's law we can express AND using OR and NOT. Let t be a test and p a program. IF t THEN p FI is equivalent to

```
fresh-aux := NIL ;
WHILE AND( fresh-aux = NIL , t ) DO
      p ; fresh-aux := 'a' OD
```

With this, IF t THEN p1 ELSE p2 FI is equivalent to

"copy the variables in t into auxiliary ones let t-aux be the same test as t but performed on the corresponding auxiliary variables"

```
IF t-aux THEN p1 FI ; IF NOT( t-aux ) THEN p2 FI
```

Using `IF - THEN - ELSE - FI`, one can express a program that is equivalent to a `CASE` statement. We are ready to write the equivalent of `x := HEAD( w )` which takes the first character of a nonempty string, and does nothing if the string is empty. For simplicity we assume that we have only two characters, `'a'` and `'b'`.

```
CASE  w = CONS( 'a' , TAIL( w ) ) => x := 'a' ;
      w = CONS( 'b' , TAIL( w ) ) => x := 'b' ;
      DEFAULT                     => SKIP
ESAC
```

As an exercise, try to write a program that reverses a string (the reverse of `'ababb'` is `'bbaba'`).

Definition 1.3 *Fix two alphabets, an "input" alphabet Σ_i, and an "output" alphabet Σ_o. A partial function $f : \Sigma_i^* \dot{\to} \Sigma_o^*$ is said to be* **WP-computable** *if there exists a while-program P with two distinguished variables, called* `input` *and* `output`*, such that for any $u \in \Sigma_i^*$, P halts when started with u in* `input` *iff f is defined on u, and when P halts, it has $v = f(u)$ in* `output`*. In such a situation, we say that P* computes *f.*

Theorem 1.4
A partial function is Turing-computable iff it is WP-computable.

This result can be proved rigorously but I feel that such proofs are not terribly informative. For example, once we know about multitape Turing machines, we can simulate a while program with n variables (identifiers) by a Turing machine with n+k tapes where we always maintain the content of the variables on the first n tapes and we used then other k to compute the values of expressions and test. Thus, with knowledge of a couple of Turing machine "programming tricks" which go some way toward making available the usual tools of Pascal-like programming, I propose that we rely on our programming experience to convince ourselves that the remaining details can be tediously but straightforwardly supplied.

1.3 The Church-Turing Thesis

The Turing Machine model of computation is quite simple, and it is a remarkable fact that it has the same "computing power" as, say, Pascal programs. Here I have in mind *idealized* Pascal programs, in particular ones in which the variables can hold data of any size. In reality, there is always an upper bound on, say, the length of a string that a variable can hold, so we couldn't even consider computing functions that take *any* string as argument. In principle, one can rigorously prove that such Pascal programs compute the same class of partial functions as the Turing machines. In practice, there are a few proofs of equivalence between Turing machines and programs similar to the ones in certain Pascal fragments, such as the while-programs described in the previous section. Moreover, there is nothing special about Pascal here, as there are models of computation suggesting assembly languages or bare machine languages, or even other high-level programming languages, and all have been shown to be equivalent to Turing machines.

Quite a few other models of computation have been proposed by mathematicians in the last 60 years, especially at the beginnings of Computability Theory. Here are some of the more famous ones:

- General recursion equations (Herbrand-Gödel-Kleene).
- The lambda calculus (Church-Kleene-Rosser).
- μ-recursive functions (Kleene).
- Canonical deduction systems (Post).
- Normal algorithms (Markov).
- Register machines (counter/pointer/random access machines). These are various mathematical abstractions of actual (simple) computer architectures and were studied by many authors (Péter, Lambek, Minsky, Shepherdson and Sturgis, etc.)

Each of these models constitutes an attempt to capture the informal and intuitive idea of effective computability. These models of computation are quite different, and hence it is a very remarkable fact that they, and all other formalisms that have been proposed so far, have all been shown to be equivalent. This constitutes strong evidence that all and each of them has been successful in capturing intuitive computability. While the equivalences between models are formal theorems that have been *proved*, the statement that, say, Turing-computable functions are exactly the intuitive effectively computable functions is not a theorem, since it involves an intuitive concept. This statement is usually called *Church-Turing thesis* (because Church and Turing made such statements independently about the formalisms they proposed—λ-definability and computability by Turing machines):

> **Thesis.** *The intuitively and informally understood class of effectively computable partial functions coincides with the formally defined class of, say, Turing-computable functions.*

(In view of the equivalence results mentioned above, any of those models of computation could have been used in the formulation of the thesis instead of Turing machines.)
What is the evidence for the Church-Turing thesis? First, of course, the equivalence between the models of computation that have been proposed. Second, one "half" of the thesis is intuitively obvious: each function that is WP-computable or Turing computable is clearly effectively computable, for any reasonable meaning of the latter. Third, very, very many tasks that were claimed to be intuitively effectively computable have been, sure enough, shown to be "programmable" in some model of computation (and hence in all). Finally, there is no known task that would be acknowledged by a significant number of mathematicians, logicians, or computer scientists as being intuitively effectively computable but which has been formally shown *not* to be programmable in the models of computation.

Not surprisingly, most mathematicians, logicians, and computer scientists believe the Church-Turing thesis. I am going to rely, philosophically, on the Church-Turing thesis in the rest of these notes, by casting machine descriptions in terms of an informal and somewhat intuitive concept of "program" or "algorithm" (we are talking here about an intuition shared mostly by programming connoisseurs). This is, however, a rather "soft" dependence, since I expect the reader, making extensive use of her programming experience, and at the cost of an immense amount of work on tedious details, to be able, if challenged, to provide a fully formal version of the definitions, theorems, and proofs in terms of, say, Turing machines or while-programs. No such challenge wil be made, but the reader should be convinced that he can, in principle, meet it.

2 Computable Functions, R.E. Languages, Decidable Predicates

These are the main "characters" in the portion of the Computability Theory that we are studying. We will give a sequence of theorems that relate these concepts.
We have already defined what Turing-computable partial functions are. In view of our discussion in the previous section, we will drop the "Turing" prefix and talk about *computable* functions. The computable partial functions are also called *partial recursive functions*, and another name for Computablity Theory is Recursive Function Theory or simply Recursion Theory. In Computer Science, where *how* we program given tasks is as important as whether we *can* program them, it would be nice to reserve "recursive" for a special way of computing (by recursive procedures).
Before proceeding further, we will slightly change the definition of when a word is accpted by a Turing machine, replacing it with an equivalent one, which can also be generalized to while-programs.

Definition 2.1 *Fix an alphabet,* Σ. *A language* $L \subseteq \Sigma^*$ *is said to be* **Turing-acceptable** *if there exists a Turing machine* M *such that for any* $w \in \Sigma^*$, M *halts on input* w *iff* $w \in L$. *In such a situation, we say that* M *is an* acceptor *for* L.

Similarly, one can define acceptable languages for while-programs, etc. All these definitions yield the same class of languages. It will turn out that there is a substantially different definition, that of r.e. languages, that yields the same class, but for the moment let us call the Turing-acceptable languages—*halt-acceptable*. We can immediately give a characterization of such languages in terms of the computability of certain partial functions.
A bit of notation: for any language $L \subseteq \Sigma^*$ consider the partial function $f_L : \Sigma^* \to \Sigma^*$ given by

$$f_L(u) \stackrel{\text{def}}{=} \begin{cases} \epsilon & \text{if } u \in L \\ \text{undefined} & \text{if } u \notin L \end{cases}$$

Theorem 2.2
A language L *is halt-acceptable iff the corresponding partial function* f_L *is computable.*

Proof. First, assume that L is halt-acceptable. Let M be an acceptor for L. Construct a TM M' that computes f_L as follows:
M': "on input u
run M on u
if it halts then output ϵ and halt"
Indeed, M' halts on input u iff M halts on u iff $u \in L$ iff f_L is defined on u. Moreover, when M' halts on input u, it outputs $\epsilon = f_L(u)$. Hence, M' computes f_L.
Conversely, suppose that f_L is computed by some machine M''. Then, M'' is also an acceptor for L since M'' halts on input u iff f_L is defined on u iff $u \in L$. **End of proof.**
For the next definition, we will need Turing machines that are able to *print* strings. We can do this using multi-tape machines and the "programming" convention that the "printed" strings are written on a special tape, in sequence, from left to right, separated by some delimiter.
Note. In the case of while-programs just add a new clause to the definition of statements:

$$< \text{statement} > ::= \cdots \mid \texttt{PRINT}(< \text{expression} >)$$

Definition 2.3 *Fix an alphabet, Σ. A language $L \subseteq \Sigma^*$ is said to be* **Turing-enumerable** *if there exists a Turing machine E such that for any $w \in \Sigma^*$, E, starting, eventually prints w iff $w \in L$. In other words, E eventually prints all the strings in L, and only those. In such a situation, we say that E is an* enumerator *for L.*

Note that E may run forever, printing an infinite language. It may also run forever without printing, after having printed a finite set of strings. Note also that we did not require that a given string be printed at most once. This, however, can be achieved by a machine that also stores all the strings it prints and makes sure each candidate for printing is not already stored.
As before, we drop the "Turing" prefix, since equivalent notions are obtained by using any of the known models of computation. We shall call such languages *recursively enumerable.* As promised:

Theorem 2.4
A language is halt-acceptable iff it is recursively enumerable.

Proof. First, let L be a recursively enumerable and let us show that L is halt-acceptable. Let E be an enumerator for L. Construct an acceptor for L as follows:
M: "on input w
run E, and each time E prints a string u
    compare u with w
    if $u = w$, halt
if not, continue running E
if E halts before a string that equals w is found, loop"
Clearly, M halts on w iff E eventually prints w iff $w \in L$.
For the converse, we need a little preparation. First, we need a total ordering on strings such that we can write a subroutine which given a positive integer n will return the n'th string in the

ordering. We define the *canonical ordering* on strings to be the order that is given, first by the length of the string, and then, among strings of equal length, by lexicographic (dictionary) order. We will also need a programming technique for generating all possible pairs of positive integers. (This can be done in various ways. For example, it is easy to write a program that generates the sequence (1,1),(1,2),(2,1),(3,1),(2,2),(1,3),(1,4),(2,3), *etc.*)

With this, let L be halt-acceptable and let M be an acceptor for it. Construct an enumerator for L as follows:

E: "generate all pairs of positive integers

for each pair (i, j),

compute the i'th string in the canonical ordering, call it w

run M on input w for up to j steps

if this halts, print w

repeat with the next pair (i, j)"

Indeed, w is eventually printed by E iff there exists a pair of positive integers (i, j) such that w is the i'th string in the canonical ordering and M halts on w in up to j steps iff w is a string on which M halts iff $w \in L$. **End of proof.**

The terminology "halt-acceptable" was temporarily invented for these notes. From now on we shall call such languages *recursively enumerable*, or simply *r.e.*, and, in view of the previous theorem, describe them either by acceptors or by enumerators, whichever is more convenient in specific proofs. One can sometimes find r.e. languages under the names *semi-decidable* or *semi-recursive*.

Back to the relationship between r.e. languages and computable functions. While Theorem 2.2 states that halt-acceptability (r.e.-ness) can be characterized in terms of computable partial functions, the next theorem is stating a sort of converse.

First another bit of notation: if $f : \Sigma_i^* \stackrel{\cdot}{\to} \Sigma_o^*$ is a partial function, then

$$\text{graph}(f) \stackrel{\text{def}}{=} \{\langle u, v\rangle \mid u \in \Sigma_i^*, v \in \Sigma_o^*, f \text{ is defined on } u, \text{ and } f(u) = v\}$$

Here, $\langle u, v\rangle$ is a pair of strings. Pairs of strings are finite objects and can be encoded as strings too, so that we can compute with them in our general framework. For example, we can assume that $\langle$, the comma, and $\rangle$ do not belong to $\Sigma_i \cup \Sigma_o$, literally think of $\langle u, v\rangle$ as a string over the alphabet $\Sigma_i \cup \Sigma_o \cup \{\langle, \text{comma}, \rangle\}$, and hence think of graph(f) as a language over this alphabet.

Theorem 2.5

A partial function f is computable iff the corresponding language graph(f) *is r.e.*

Proof. Assume f is computable and show graph(f) is r.e. Let M be a TM that computes f. Construct an acceptor for graph(f) as follows:

M': "on input w

if $w = \langle u, v\rangle$ for some u, v

then run M on u

if it halts, compare the output with v

if they are equal, halt
otherwise loop
if $w \neq \langle u, v\rangle$ for any u, v,
then loop"
Indeed, M' halts on w iff $w = \langle u, v\rangle$ for some u, v and M halts on input u with output v iff $w = \langle u, v\rangle$ for some u, v, f is defined on u, and $f(u) = v$ iff $w \in \text{graph}(f)$.
Conversely, assume that graph(f) is r.e. and let's show that f is computable. Let E be an enumerator for graph(f). Construct a TM that computes f as follows;
M: "on input u
run E, and each time E prints a string w
if there is a v such that $w = \langle u, v\rangle$
then output v and halt
otherwise continue running E
if E halts before such a w is printed, loop"
Clearly, M halts on input u iff E eventually prints $\langle u, v\rangle$ for some v iff there is a v such that $\langle u, v\rangle \in \text{graph}(f)$ iff f is defined on u. Moreover, when M halts on input u, it outputs a v such that $\langle u, v\rangle \in \text{graph}(f)$ hence it outputs $v = f(u)$. **End of proof.**
In view of Theorems 2.2 and 2.5, we can say that the concepts of computable partial function and r.e. language are very strongly related.

Definition 2.6 *Fix an alphabet, Σ. A language $L \subseteq \Sigma^*$ is said to be* **Turing-decidable** *if there exists a Turing machine D that halts on all inputs, and such that for any $w \in \Sigma^*$, when D halts on input w it outputs Ⓨ if $w \in L$ and Ⓝ if $w \notin L$. In such a situation, we say that D is a* decider *for L.*

Decidable languages are called *recursive* in some textbooks. The reader may have noticed the "decidable predicates" in the title of this section, while we have just defined "decidable languages". The reason is a perfect one-to-one correspondence (bijection) between decidable languages and certain computable total functions called *predicates*.
Let $\mathbb{B} \stackrel{\text{def}}{=} \{Ⓨ, Ⓝ\}$ A *predicate* on Σ^* is a *total* function from Σ^* to $\mathbb{B}$. We can think of $\mathbb{B}$ as a subset of some Σ_o^*, for example if Σ_o actually contains Ⓨ and Ⓝ. This way, a predicate is a function taking strings to strings so we can ask whether it is computable. Note however that predicates are quite special among general partial functions, being total and very restricted in their possible outputs.
There is an obvious and well-known bijection between the predicates on Σ^* and the subsets of Σ^*. In one direction, this bijection associates to a predicate $p : \Sigma^* \to \mathbb{B}$ its *yes-language* (Yes-languages are usually called "truth-sets"):

$$Y_p \stackrel{\text{def}}{=} \{w \in \Sigma^* \mid p(w) = Ⓨ\}$$

In the other direction, this bijection associates to a language $L \subseteq \Sigma^*$ its *characteristic predicate*

$\chi_L : \Sigma^* \to \mathbb{B}$ defined by

$$\chi_L(w) \stackrel{\text{def}}{=} \begin{cases} \text{Ⓨ} & \text{if } w \in L \\ \text{Ⓝ} & \text{if } w \notin L \end{cases}$$

The next theorem says that this bijection is maintained when we restrict ourselves to decidable languages and computable predicates.

Theorem 2.7

A language is decidable iff its characteristic predicate is computable. A predicate is computable iff its yes-language (or truth-set) is decidable.

Proof. It is easy to see that if D is a decider for L then D is also a TM that computes χ_L. Conversely, it is just as easy to see that if M computes a predicate p then M is also a decider for Y_p. **End of proof.**

In view of this result, and of its proof, we can use "decidable language" interchangeably with "computable predicate". In fact, people even use, with a slight abuse of terminology, "decidable predicate", hence the title of this section.

Thus, while r.e. languages are strongly related to partial computable functions, decidable languages are strongly related to total computable functions. In fact, one can even show an analogue of Theorem 2.5: a total function is computable iff its graph is decidable. Just as total functions are particular cases of partial functions, *any decidable language is r.e.* To see this, note that any decider for a language L can be transformed into an acceptor for L by simply replacing "output Ⓝ then halt" by "loop".

We will see in the next section that the converse fails: there are r.e. languages which are not decidable. Of course, in Computer Science we are first interested in decidability, whenever possible. Thus we end this section on a positive note with a criterion for showing decidability.

First, note that *the class of decidable languages is closed under complementation*: any decider D for L can be transformed into a decider for its complement $\overline{L}$ by swapping Ⓨ and Ⓝ. The same is no true for r.e. languages. This will follow from the existence of r.e. but undecidable languages and the next theorem. First, a widely used terminology:

Definition 2.8 *Let Σ be a finite alphabet. A set $L \subseteq \Sigma^*$ is said to be* co-r.e. *if its complement $\overline{L} \stackrel{\text{def}}{=} \Sigma^* - L$ is r.e.*

Theorem 2.9

A language is decidable iff it is both r.e. and co-r.e.

Proof. If a language is decidable, so is its complement hence the language is both r.e. and co-r.e. Conversely, let L be both r.e. and co-r.e. Let M be a acceptor for L and $\overline{M}$ an acceptor for $\overline{L}$. Construct a decider for L as follows:

D: "on input w

run M and $\overline{M}$ both on input w, "in parallel" (interleaving the computation)

if M halts, output Ⓨ and halt
if $\overline{M}$ halts, output Ⓝ and halt"
Indeed, D must halt on all inputs because any input is either in L or in $\overline{L}$ so one of M , $\overline{M}$ must halt, and in both cases D halts. Moreover, when D halts on w, it outputs Ⓨ if M halts on w, that is, if $w \in L$, and it outputs Ⓝ if $\overline{M}$ halts on w, that is, if $w \in \overline{L}$, equivalently $w \notin L$. It follows that D is a decider for L. **End of proof.**
Finally, another important connection between r.e.-ness and decidability.

Theorem 2.10
A language L is r.e. iff there exists a decidable language R such that for any $u \in \Sigma^*$

$$u \in L \text{ iff there exists } v \in \Sigma^* \text{ such that } \langle u, v \rangle \in R$$

Proof. EXERCISE! **End of proof.**

3 Undecidability

3.1 The Halting Problem

Turing machine descriptions can be encoded as strings too, and hence can be taken as inputs by other machines. If M is a TM, we shall denote by $[M]$ its description encoded as a string. We will assume that it is decidable whether an arbitrary string is a machine description or not. Clearly, encodings of TM descriptions with this property can be given. The following is a fundamental result of Computability Theory:

> **The Universal Simulation Theorem** (Gödel-Church-Kleene-Turing)
> *There exists (can be written down!) a Turing machine U which, on an input of the form $\langle [M], w \rangle$, simulates the computation of the Turing machine M on input w.*

"Proof". U is an interpreter for Turing machines. (A formal proof of this result is very tedious and, nowadays, uninformative. We rely heavily on the reader's programming experience to accept this result.)
Definition. The *Halting Problem* is the language

$$K \stackrel{\text{def}}{=} \{\langle [M], w \rangle \mid M \text{ halts on input } w\}$$

Theorem 3.1
K is r.e.

Proof. By the Universal Simulation Theorem, U accepts K. **End of proof.**
We proceed to show that K is undecidable. Define the auxiliary *Diagonal Halting Problem*:

$$K_d \stackrel{\text{def}}{=} \{[M] \mid M \text{ halts on input} [M]\}$$

Theorem 3.2
K_d is undecidable.

Proof. Suppose, toward a contradiction, that K_d is decidable. Let D_d be a decider for K_d. Consider the folowing Turing machine (called T for trouble...):
T: "on input w
if $w = [M]$ for some M
then run D_d on input $[M]$
    if D_d outputs Ⓨ then loop
    if D_d outputs Ⓝ then halt
if $w \neq [M]$ for any M then do whatever"
Clearly, for any TM M,
T halts on input $[M]$ iff
D_d outputs Ⓝ on input $[M]$ iff $[M]$ is not in K_d iff
M loops on input $[M]$.
Instantiating in the previous equivalence $M = T$ gives T halts on input T iff T loops on input T, a contradiction. **End of proof.**

Theorem 3.3
K is undecidable.

Proof. We show that K decidable implies K_d decidable, hence we can conclude that K is undecidable from the fact that K_d is undecidable.
Let D decide K. Construct a decider for K_d as follows:
D_d: "on input w
construct $\langle w, w\rangle$ and run D on it"
Clearly D_d halts on all inputs and outputs either Ⓨ or Ⓝ, since D does so. Moreover, D_d outputs Ⓨ on input w iff D outputs Ⓨ on $\langle w, w\rangle$ iff $w = [M]$ for some M and M halts on $[M]$ iff $w \in K_d$. **End of proof.**

3.2 Many-One Reducibility

The argument we used to show that K decidable implies K_d decidable is a particular instance of a general technique called *many-one reducibility.*

Definition 3.4 *Let $L_1 \subseteq \Sigma_1^*$ and $L_2 \subseteq \Sigma_2^*$ be two sets. We say that L_1 is* **many-one reducible** *to L_2, notation $L_1 \leq_m L_2$ whenever there exists a total computable function $f : \Sigma_1^* \to \Sigma_2^*$ such that*

$$\forall w \in \Sigma_1^*, \quad w \in L_1 \text{ iff } f(w) \in L_2$$

The adjective "many-one" was introduced to distinguish this kind of reduction from others studied in Computability Theory (notably Turing-reduction). This being the only kind of reduction studied here, we will drop the "many-one" as well as the $_m$ from $\leq_m$.

It is easy to check that $K_d \leq K$ by the total computable function $f(w) = \langle w, w \rangle$. We generalize the argument in the proof of Theorem 3.3:

Theorem 3.5
If $L_1 \leq L_2$ and L_2 is decidable, then L_1 is decidable.

Proof. Let f be the total computable function performing the reduction and M a TM that computes it.
Let D_2 be a decider for L_2. Construct a decider for L_1 as follows:
D_1: "on input u
run M on u producing output v
then run D_2 on v"
D_1 halts on all inputs because M halts on all inputs (since it computes the *total* function f) and then because D_2 halts on all inputs. D_1 outputs either Ⓨ or Ⓝ, since D_2 does so. Finally, D_1 outputs Ⓨ on input u iff D_2 outputs Ⓨ on $v = f(u)$ iff $f(u) \in L_2$ iff $u \in L_1$ (where the last equivalence is by the fundamental property of the reduction). **End of proof.**
Here are two other important properties of reducibility

Theorem 3.6

1. *If $L_1 \leq L_2$ and $L_2 \leq L_3$ then $L_1 \leq L_3$.*
2. *$L_1 \leq L_2$ iff $\overline{L_1} \leq \overline{L_2}$.*

Proof. (Sketch) Part1: take the composition of the functions performing the reductions. Part 2: the same function performs the reduction between the complements too. Exercise: fill in the details! **End of proof.**
We apply the reducibility technique to show the undecidability of several questions, as a consequence of the undecidability of K. Define

$$K_\epsilon \stackrel{\text{def}}{=} \{[M] \mid M \text{ halts on input } \epsilon\}$$

$$K_{w_0} \stackrel{\text{def}}{=} \{[M] \mid M \text{ halts on input } w_0\}$$

where w_0 is a fixed arbitrary string.

$$NONEMPTY \stackrel{\text{def}}{=} \{[M] \mid M \text{ halts on } \textit{some} \text{ input}\}$$

$$TOTAL \stackrel{\text{def}}{=} \{[M] \mid M \text{ halts on } \textit{all} \text{ inputs}\}$$

Theorem 3.7
$K \leq L$ for each L in $\{K_\epsilon, K_{w_0}, NONEMPTY, TOTAL\}$.

Proof. We have grouped these languages together because the reduction is the same. Define a total function f as follows. If $u = \langle [M], w\rangle$ for some M, w then $f(u) \stackrel{\text{def}}{=} [M']$ where
M': "on input v
ignore v and run M on w"
If $u \neq \langle [M], w\rangle$ for any M, w then $f(u) \stackrel{\text{def}}{=}$ whatever, provided it is not the encoding of a TM description (that is, it is not of the form $[M'']$ for any TM M'').
We claim that f is computable. Indeed, the TM that computes f will work as compilers do, taking as input a "program text" $[M]$ (and a "manifest data" string w) and producing as output another "program text". All it will need to do is to produce a bit of TM "code" that writes w in the input area (overwriting v) and then add the TM "code" of M.
The fact that f performs the four claimed reductions follows immediately from the following observations:
1) If M halts on w then M' halts on all inputs.
2) If M' halts on some input then M halts on w.
(By the way, f also performs the reduction $K \leq K_d$.) **End of proof.**

3.3 K is r.e.-complete

We are about to see that K is quite special among the r.e. languages. The result can be interpreted as saying that K is "most" undecidable among all r.e. languages.

Definition 3.8 *A language C is said to be* **r.e. complete** *if C is r.e. and any r.e. language L is reducible to C, $L \leq C$.*

Strictly speaking, we should have said "r.e.-complete with respect to many-one reducibility", but since this is the only kind of reducibility we talk about here, we will omit the qualification.
By Theorem 3.5, if an r.e.-complete language is decidable then *all* r.e. languages are decidable. Of course, we have seen that this is not the case, but based on this observation, we can say that r.e.-complete languages are "most" undecidable among all r.e. languages.

Theorem 3.9
K is r.e.-complete.

Proof. K is r.e. by Theorem 3.1. Let L be any r.e. language. Then, there exists an acceptor for L, call it M. Consider the total function f defined by $f(w) \stackrel{\text{def}}{=} \langle [M], w\rangle$. This function is obviously computable. Finally, for any w, $w \in L$ iff M halts on w iff $\langle [M], w\rangle \in K$ iff $f(w) \in K$, hence f establishes the desired reduction. **End of proof.**

3.4 Rice's Theorem

The facts that K_ϵ, K_{w_0}, $NONEMPTY$, and $TOTAL$ are undecidable are particular aspects of a much more general negative phenomenon. These four problems can all be described in terms of properties of the r.e. language (halt-)accepted by a TM.

Definition. $\mathcal{L}(M) \stackrel{\text{def}}{=} \{w \in \Sigma^* \mid M \text{ halts on input } w\}$

To see how K_ϵ, K_{w_0}, $NONEMPTY$, and $TOTAL$ can be seen as particular cases of a more general problem note that:

$$K_\epsilon = \{[M] \mid \mathcal{L}(M) \ni nil\}$$

$$K_{w_0} = \{[M] \mid \mathcal{L}(M) \ni w_0\}$$

$$NONEMPTY = \{[M] \mid \mathcal{L}(M) \neq \emptyset\}$$

$$TOTAL = \{[M] \mid \mathcal{L}(M) = \Sigma^*\}$$

All languages of the form $\mathcal{L}(M)$ are r.e. Let $\mathcal{RE}$ be the set of all r.e. languages over our fixed alphabet. We define a *property of r.e. languages* to be simply a function defined on the set of r.e. languages taking truth values $\pi : \mathcal{RE} \longrightarrow \mathbb{B}$. Instead of $\pi(L) = \texttt{'YES'}$ we say that "L has property π". We will consider the following problem associated to a property of r.e. languages π:

$$J_\pi \stackrel{\text{def}}{=} \{[M] \mid \mathcal{L}(M) \text{ has property } \pi\}$$

Hence, K_ϵ, K_{w_0}, $NONEMPTY$, and $TOTAL$ are all J_π's, where π is, respectively, "contains ϵ", "contains w_0", "does not equal the empty set", and "equals Σ^*".

A property of r.e. languages is said to be *non-trivial* whenever there is an r.e. language that has it and there is also an r.e. language that does not have it. There are only two trivial properties: the one that all r.e. have and the one that no r.e. language has. In both cases the associated set J_π is trivially decidable. Note that all four properties mentioned above are non-trivial.

Rice's Theorem

For any nontrivial property of r.e. languages π, the set J_π is undecidable

Proof. Let π be non-trivial. We distinguish two cases:

Case 1: $\emptyset$ does not have property π.

Since π is non-trivial, there exists an r.e. language L that has the property π. Let A be a TM that accepts L.

We show that $K \leq J_\pi$. Define a total function f as follows. If $u \equiv \langle [M], w\rangle$ for some M, w then $f(u) \stackrel{\text{def}}{=} [M']$ where

M':

"on input v

save v then run M on w

if it halts, run A on v"

If $u \not\equiv \langle [M], w\rangle$ for any M, w then $f(u) \stackrel{\text{def}}{=} \epsilon$. f is clearly computable. To see that it performs the desired reduction note that if M halts on w, then M' halts on v iff A halts on v, hence

$\mathcal{L}(M') = \mathcal{L}(A) = L$ which has property π. For the converse, we check its contrapositive: if M loops on w then M' loops on all inputs hence $\mathcal{L}(M') = \emptyset$ which does not have property π, by the assumption in this case.

Case 2: $\emptyset$ has property π.

Consider the property $\neg\pi$ which is true of exactly those languages which do *not* have property π. $\emptyset$ does not have property $\neg\pi$ so by *Case 1* we have $K \leq J_{\neq\pi}$. By Theorem 3.6, $\overline{K} \leq \overline{J_{\neg\pi}}$.

Let N be the set of all strings that are *not* TM description encodings. It is easy to see that $\overline{J_{\neg\pi}} = J_\pi \cup N$. We claim that $\overline{J_{\neg\pi}} \leq J_\pi$. Indeed, this reduction is given by the total computable function f defined by $f([M]) \stackrel{\text{def}}{=} [M]$ and $f(w) \stackrel{\text{def}}{=} [LOOP]$ for $w \in N$ where $LOOP$ is a TM that loops on all inputs ($\mathcal{L}(LOOP) = \emptyset$ which has property π hence $[LOOP] \in J_\pi$).

By transitivity of reducibility (Theorem 3.6), we conclude that $\overline{K} \leq J_\pi$. Since $\overline{K}$ is undecidable, so is J_π. **End of proof.**

In fact, the proof we gave yields a stronger result than the one stated. Because $\overline{K}$ is not even r.e., J_π cannot be even r.e. in *Case 2*. We obtain

> **Sharpened Rice's Theorem**
> *For any nontrivial property of r.e. languages π, the set J_π is undecidable. Moreover, if $\emptyset$ has property π then J_π is not even r.e.*

As a consequence, we obtain that K_ϵ, K_{w_0}, $NONEMPTY$, and $TOTAL$ are undecidable. Other interesting consequences:

$$NON - K_\epsilon \stackrel{\text{def}}{=} \{[M] \mid M \text{ loops on } \epsilon \} \text{ is not r.e.}$$

$$NON - K_{w_0} \stackrel{\text{def}}{=} \{[M] \mid M \text{ loops on } w_0\} \text{ is not r.e.}$$

$$EMPTY \stackrel{\text{def}}{=} \{[M] \mid M \text{ loops on all inputs } \} \text{ is not r.e.}$$

$$NONTOTAL \stackrel{\text{def}}{=} \{[M] \mid M \text{ loops on some input } \} \text{ is not r.e.}$$

$$FIN \stackrel{\text{def}}{=} \{[M] \mid M \text{ halts on finitely many inputs } \} \text{ is not r.e.}$$

$$REG \stackrel{\text{def}}{=} \{[M] \mid \mathcal{L}(M) \text{ is regular } \} \text{ is not r.e.}$$

$$CF \stackrel{\text{def}}{=} \{[M] \mid \mathcal{L}(M) \text{ is context-free } \} \text{ is not r.e.}$$

$$DEC \stackrel{\text{def}}{=} \{[M] \mid \mathcal{L}(M) \text{ is decidable } \} \text{ is not r.e.}$$

3.5 Beyond R.E. and Co-R.E.

Are there languages which are neither r.e. nor co-r.e.? It turns out that $TOTAL$ is such a language. From the Sharpened Rice's Theorem we know that $NONTOTAL$ is not r.e. An argument in the spirit of that which showed $\overline{J_{\neg\pi}} \leq J_\pi$ in the proof of Rice's Theorem will show that $NONTOTAL \leq \overline{TOTAL}$. Hence, $\overline{TOTAL}$ cannot be r.e. either, from which follows that $TOTAL$ is not co-r.e. It remains to show that $TOTAL$ is not r.e. We will do so by proving that $\overline{K} \leq TOTAL$. By Theorem 3.6, this is equivalent to $K \leq \overline{TOTAL}$. Since $NONTOTAL \leq \overline{TOTAL}$, it is sufficient to show $K \leq NONTOTAL$. To see this, consider the total function f defined as follows. If $u \equiv \langle [M], w\rangle$ for some M, w then $f(u) \stackrel{\text{def}}{=} [M']$ where
M':
"on input v
run M on w for length(v) steps
if it halts, loop
otherwise halt"
If $u \not\equiv \langle [M], w\rangle$ for any M, w then $f(u) \stackrel{\text{def}}{=} \epsilon$. Clearly f is computable. Moreover, M halts on w iff there is an n such that M halts on w in n steps iff there is a v such that M halts on w in length(v) steps iff there is a v such that M' loops on v. This completes the argument.

4 Undecidability of String Rewriting

The String Rewriting Problem (SRP) is an example of an undecidable problem that does not seem to talk directly about programs (while all the undecidable problems we've encountered before do). It is a generalization of many other problems that occur in symbol manipulation languages or in automated theorem proving.
Fix a finite alphabet Σ. A *string rewrite rule*, r, is just an ordered pair of strings over Σ, .e. $r \equiv \langle w_1, w_2\rangle$. We will use the more suggestive notation $r : w_1 \longrightarrow w_2$. Let R be a finite set of string rewrite rules:

$$\begin{array}{lcll} r_1 & : & u_1 & \longrightarrow & v_1 \\ r_2 & : & u_2 & \longrightarrow & v_2 \\ & & \vdots & & \\ r_n & : & u_n & \longrightarrow & v_n \end{array}$$

For each rule $r_i \in R$, define a binary relation $\xrightarrow{r_i}$ on strings over Σ as follows (one-step rewriting according to the rule r_i)

$$\forall u, v \in \Sigma^*, \quad u \xrightarrow{r_i} v \quad \stackrel{\text{def}}{\iff} \quad \exists x, y \in \Sigma^*, \ u \equiv x u_i y \ \wedge \ v \equiv x v_i y$$

and then a binary relation $\xrightarrow{R}$ (rewriting according to the set of rules R)

$$\forall u, v \in \Sigma^*, \quad u \xrightarrow{R} v \quad \stackrel{\text{def}}{\iff} \quad \exists i, \ u \xrightarrow{r_i} v$$

Let $\overset{R}{\longrightarrow\!\!\!\!\!\rightarrow}$ be the transitive-reflexive closure of $\overset{R}{\longrightarrow}$. In other words

$$u \overset{R}{\longrightarrow\!\!\!\!\!\rightarrow} v \quad \iff \quad \exists k \geq 0,\ \exists x_0, x_1, \ldots, x_k \in \Sigma^*,\ \ u \equiv x_0 \overset{R}{\longrightarrow} x_1 \overset{R}{\longrightarrow} \cdots \overset{R}{\longrightarrow} x_k \equiv v$$

so we call this relation, rewriting in 0 or more steps according to the set of rules R, or simply, rewriting.

The String Rewriting Problem is the following: given a finite set of string rewrite rules R and two strings u and v, does u rewrite to v according to R? We will see that this is undecidable. In other words, we will see that the set

$$SRP \overset{\text{def}}{=} \{\langle R, u, v\rangle \mid u \overset{R}{\longrightarrow\!\!\!\!\!\rightarrow} v\}$$

is not decidable.

Note that we insist on rewriting in 0 or more steps. In fact, it is not hard to see that one-step rewriting is decidable. To decide whether $u \overset{R}{\longrightarrow} v$, simply check for each rule r_i and each substring u' of u whether $u' \equiv u_i$. If so, rewrite u according to r_i and check whether the result is v.

Theorem 4.1

SRP is r.e.

Proof. By the considerations in the previous paragraph, given $k, x_0, x_1, \ldots, x_k, u, v$, it is decidable whether $u \equiv x_0 \overset{R}{\longrightarrow} x_1 \overset{R}{\longrightarrow} \cdots \overset{R}{\longrightarrow} x_k \equiv v$. Using this, we conclude that *SRP* is r.e. by Theorem 2.10. **End of proof.**

However,

Theorem 4.2

SRP is undecidable.

Proof. We prove this by showing that $K \leq SRP$. It turns out that we need to be quite specific about the model of computation in order to encode the Halting Problem into the String Rewriting Problem and that it is convenient to choose the Turing Machine model for this.

We will describe the total computable function that performs the reduction $K \leq SRP$ by the FC-program that computes it. This program will take an input of the form $\langle [M], w\rangle$ where $[M]$ is a description of a Turing Machine M and w is an arbitrary input for M and produce an output of the form $\langle R, u, v\rangle$. In order for this to be a many-one reduction, it will have to be the case that M halts on w iff $u \overset{R}{\longrightarrow\!\!\!\!\!\rightarrow} v$.

Without loss of generality, we will assume that M never *writes* blank, B, and that it has just *one* halting state, h. Let Γ be the tape alphabet of M and Q its set of states. Then, R will consist of rules for rewriting strings over the alphabet $\Sigma \overset{\text{def}}{=} \Gamma \cup Q \cup \{\$\}$ where \$ is a fresh auxiliary symbol (the need for \$ will explained below). Recall that an *instantaneous description* (or *configuration*) of M is a string $\alpha q \beta$ over $\Gamma \cup Q$ where $q \in Q$ is the current state of the finite control, and $\alpha\beta \in \Gamma^*$ is the content of the tape up to the rightmost nonblank symbol such that the tape head is scanning

the leftmost symbol of β, or, if $\beta = \text{nil}$, the head is scanning a blank. (Since the machine is not writing blanks, this is somewhat simpler than the earlier definition.) We will try to simulate the moves of the Turing machine by one-step rewritings on strings representing configurations. Let q_0 be the start state of M. The reduction will construct

$$u \stackrel{\text{def}}{=} q_0 w\$ \qquad \text{and} \qquad v \stackrel{\text{def}}{=} h\$$$

which correspond roughly to an initial and a final configuration of a halting computation of M on input w. (Except for the presence of \$ and the absence of the tape content in the final configuration.) The reduction will also construct a finite set R of string rewrite rules, as follows.
For each right move,

$$RM \equiv \delta(\gamma, q) = (\gamma', q', R)$$

which can be found in $[M]$, such that $\gamma \not\equiv B$, the reduction will construct a rewrite rule

$$r_{RM} \ : \ q\gamma \longrightarrow \gamma' q'$$

since rewriting according to this rule corresponds to the efect of a move RM on some configuration of the machine:

$$\alpha q \gamma \beta' \stackrel{r_{RM}}{\longrightarrow} \alpha \gamma' q' \beta'$$

For each left move,

$$LM \equiv \delta(\gamma, q) = (\gamma', q', L)$$

which can be found in $[M]$, such that $\gamma \not\equiv B$, the reduction will construct a finite set of rewrite rules, one for each $\gamma'' \in \Gamma$

$$r_{LM}(\gamma'') \ : \ \gamma'' q \gamma \longrightarrow q' \gamma'' \gamma'$$

since a move LM on some configuration of the machine can be simulated by rewriting according to one of these rules:

$$\alpha' \gamma'' q \gamma \beta' \stackrel{r_{LM}(\gamma'')}{\longrightarrow} \alpha' q' \gamma'' \gamma' \beta'$$

You have probably noticed the restriction $\gamma \not\equiv B$. Indeed, the first time the head gets to scan a blank, the configuration will be of the form αq and it will not be possible to further rewrite this with rules whose left hand side is of the form qB. If we drop the B then we get rewrite rules whose left hand side is just q and they are applicable to many other configurations, leading to ambiguity and non-deterministic behavior in the rewriting. the solutions is to add the auxiliary symbol \$ to mark the right end of the configuration and the following rewrite rules.
For each right move reading blank,

$$RM \equiv \delta(B, q) = (\gamma', q', R)$$

the reduction will construct a rewrite rule

$$r_{RM} \ : \ q\$ \longrightarrow \gamma' q' \$$$

For each left move reading blank,

$$LM \equiv \delta(B, q) = (\gamma', q', L)$$

the reduction will construct again a finite set of rewrite rules, one for each $\gamma'' \in \Gamma$

$$r_{LM}(\gamma'') \ : \ \gamma'' q\$ \longrightarrow q'\gamma''\gamma'\$$$

Finally, we have to make sure that the rewriting process can "detect" that the halting state has been reached. Since we cannot take v to be a "pattern" like $\_h\_$, the reduction constructed $v \equiv h\$$ and therefore it will also need, for each $\gamma \in \Gamma$, to construct the rewrite rules

$$r_{hL}(\gamma) \ : \ \gamma h \longrightarrow h \qquad r_{hR}(\gamma) \ : \ h\gamma \longrightarrow h$$

Obviously, the reduction will only have to do searches through the (finite) description $[M]$ as well constructions by terminating transformations. Hence, it will compute a total function.
Moreover, if M halts on w then there is a halting computation of M on w, *i.e.,* a finite sequence of configurations starting with the initial one and ending with one containing the halt state, such that each two consecutive configurations in the sequence are related by some move of M. But then, by the discussion that accompanied the description of the reduction, $u \overset{R}{\longrightarrow\!\!\!\!\!\rightarrow} w$ where w is a string of the form $\alpha h \beta\$$ with $\alpha, \beta \in \Gamma^*$. Then $w \overset{r_{hL}, r_{hR}}{\longrightarrow\!\!\!\!\!\rightarrow} v$ hence $u \overset{R}{\longrightarrow\!\!\!\!\!\rightarrow} v$.
Conversely, if $u \overset{R}{\longrightarrow\!\!\!\!\!\rightarrow} v$, since only r_{hL} and r_{hR} have h occurring in their left hand side, there must be some w such that $u \overset{R}{\longrightarrow\!\!\!\!\!\rightarrow} w \overset{r_{hL}, r_{hR}}{\longrightarrow\!\!\!\!\!\rightarrow} v$. By induction on the length of $u \overset{R}{\longrightarrow\!\!\!\!\!\rightarrow} w$, one can then see that the strings that occur in this reduction sequence represent a sequence of configurations that constitute a halting computation of M on w. **End of proof.**