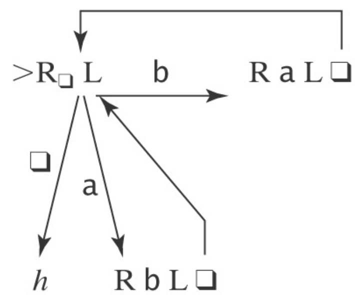
Part IV: Turing Machines and Undecidability

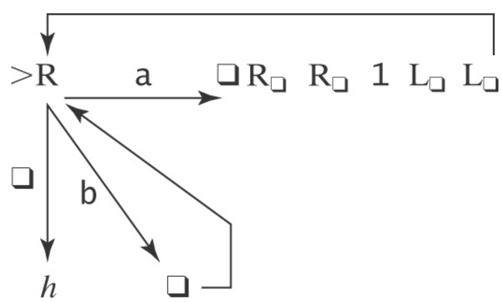
17 Turing Machines

1) Give a short English description of what each of these Turing machines does:
   a) Σ_M = {a, b}. M =



Shift the input string one character to the right and replace each b with an a and each a with a b.

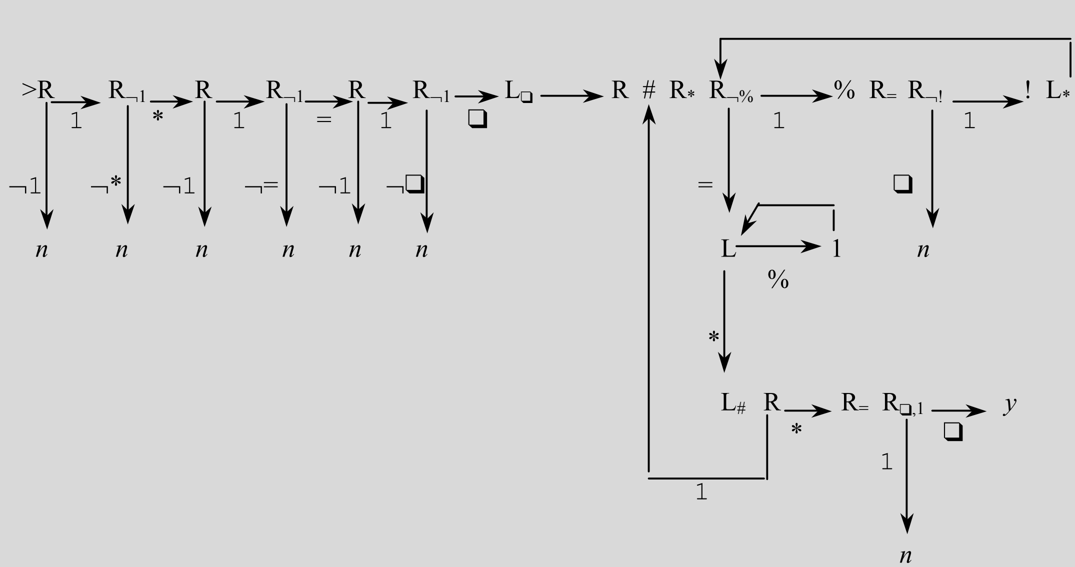
   b) Σ_M = {a, b}. M =



Erase the input string and replace it with a count, in unary, of the number of a's in the original string.

2) Construct a standard, one-tape Turing machine M to decide each of the following languages L. You may find it useful to define subroutines. Describe M in the macro language described in Section 17.1.5.
   a) $\{x * y = z : x, y, z \in 1^+$ and, when x, y, and z are viewed as unary numbers, $xy = z\}$. For example, the string 1111*11=11111111 $\in L$.

Check for legal form | Check for legal arithmetic



b) $\{a^i b^j c^i d^j, i, j \geq 0\}$.

c) $\{w \in \{a, b, c, d\}^*: \#_b(w) \geq \#_c(w) \geq \#_d(w) \geq 0\}$.

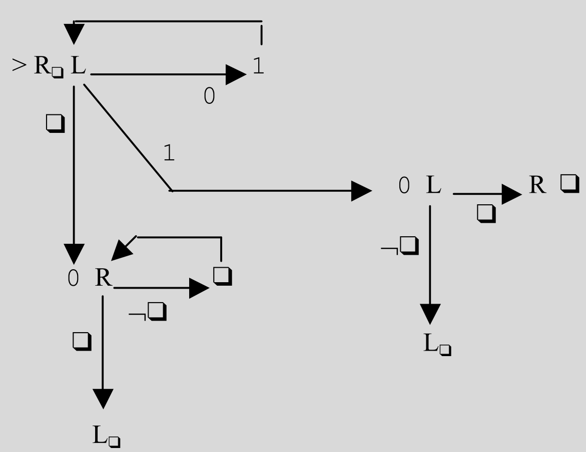
3) Construct a standard 1-tape Turing machine M to compute each of the following functions:

a) The function sub_3, which is defined as follows:

$$sub_3(n) = \begin{cases} n-3 \text{ if } n > 2 \\ 0 \text{ if } n \leq 2. \end{cases}$$

Specifically, compute sub_3 of a natural number represented in binary. For example, on input `10111`, M should output `10100`. On input `11101`, M should output `11010`. (Hint: you may want to define a subroutine.)

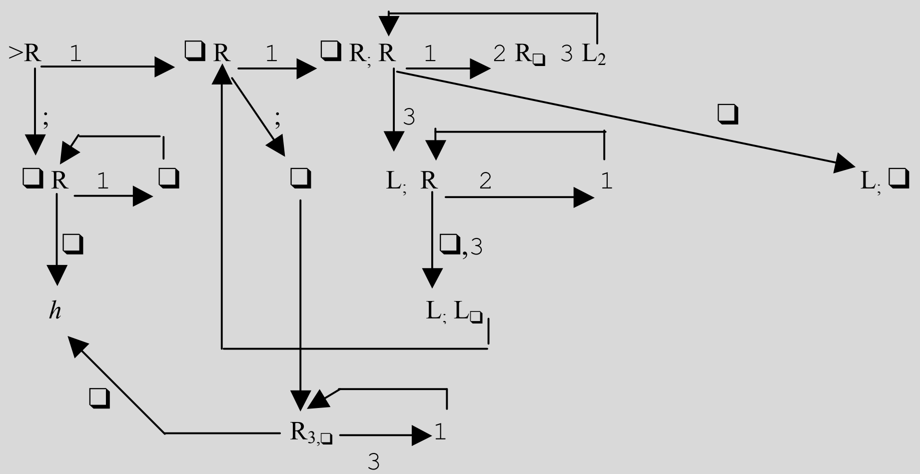
We first define a subroutine that we will call S to subtract a single 1:



Now we can define M as: > S S S

b) Addition of two binary natural numbers (as described in Example 17.13). Specifically, given the input string <x>`;`<y>, where <x> is the binary encoding of a natural number x and <y> is the binary encoding of a natural number y, M should output <z>, where z is the binary encoding of $x + y$. For example, on input `101;11`, M should output `1000`.

c) Multiplication of two unary numbers. Specifically, given the input string <x>`;`<y>, where <x> is the unary encoding of a natural number x and <y> is the unary encoding of a natural number y, M should output <z>, where z is the unary encoding of xy. For example, on input `111;1111`, M should output `111111111111`.



This machine first erases the first `1` in x. Then it uses each of the others to create a new copy of y. Each time it uses a `1` in x, it writes over it with a blank. Once all the ones in x have created their copies of y, the ; is erased.

d) The proper subtraction function *monus*, which is defined as follows:

$monus(n, m) =$	$n - m$ if $n > m$
	0 if $n \leq m$

Specifically, compute *monus* of two natural numbers represented in binary. For example, on input `101;11`, M should output `10`. On input `11;101`, M should output `0`.

4) Define a Turing Machine M that computes the function f: {`a`, `b`}* → N, where:

$f(x)$ = the unary encoding of $max(\#_{\texttt{a}}(x), \#_{\texttt{b}}(x))$.

For example, on input `aaaabb`, M should output `1111`. M may use more than one tape. It is not necessary to write the exact transition function for M. Describe it in clear English.

We can use 3 tapes: Tape 1: input
Tape 2: write `1` for every `a` in the input
Tape 3: write `1` for every `b` in the input

- Step 1: Move left to right along tape 1. If the character under the read head is `a`, write `1` on tape 2 and move one square to the right on tapes 1 and 2. If the character under the read is `b`, write `1` on tape 3 and move one square to the right on tapes 1 and 3. When tape 1 encounters a blank, go to step 2.
- Step 2: Move all three read heads all the way to the left. Scan the tapes left to right. At each step, if there is `1` on either tape 2 or 3 (or both), write `1` on tape 1. Move right on all tapes except that as soon as either tape 2 or tape 3 (but not both) gets to a blank, stop moving right on that tape and continue this

process with just the other one. As soon as the other of them (tape 2 or 3) also hits a blank, go to step 3. At this point, tape 1 contains the correct number of `1`'s, plus perhaps extra `a`'s and `b`'s that still need to be erased.

- Step 3: Scan left to right along tape 1 as long as there are `a`'s or `b`'s. Rewrite each as a blank. As soon as a blank is read, stop.

5) Construct a Turing machine *M* that converts binary numbers to their unary representations. So, specifically, on input <*w*>, where *w* is the binary encoding of a natural number *n*, *M* will output 1^n. (Hint: use more than one tape.)

M will use three tapes. It will begin by copying its input to tape 2, where it will stay, unchanged. Tape 1 will hold the answer by the end. Tape 3 will hold a working string defined below. *M* will initialize itself by copying its input to tape 2 and writing `1` on tape 3. Then it will begin scanning tape 2, starting at the rightmost symbol, which we'll call symbol 0. As *M* computes, tape 3 will contain 1^i if *M* is currently processing the i^{th} symbol (from the right, starting numbering at 0). Assume that *M* has access to a subroutine *double* that will duplicate whatever string is on tape 3. So if that string is *s*, it will become *ss*. After initialization, *M* operates as follows:

For each symbol *c* on tape 2, do:

1. If $c = 1$, then append a copy of the nonblank region of tape 3 to the end of the nonblank region of tape 1. (If this is the first append operation, just write the copy on the tape where the read/write head is.)
2. Call *double*.
3. Move the read head on tape 2 one square to the left.
4. If the square under the read/write head on tape 2 is ❑, halt. The answer will be on tape 1.

14) Encode the following Turing Machine as an input to the universal Turing machine:

$M = (K, \Sigma, \Gamma, \delta, q_0, \{h\})$, where:

$K = \{q_0, q_1, h\}$,
$\Sigma = \{$`a`, `b`$\}$,
$\Gamma = \{$`a`, `b`, `c`, ❑$\}$, and
δ is given by the following table:

q	σ	$\delta(q, \sigma)$
q_0	a	$(q_1$, b, $\rightarrow)$
q_0	b	$(q_1$, a, $\rightarrow)$
q_0	❑	$(h$, ❑ , $\rightarrow)$
q_0	c	$(q_0$, c, $\rightarrow)$
q_1	a	$(q_0$, c, $\rightarrow)$
q_1	b	$(q_0$, b, $\leftarrow)$
q_1	❑	$(q_0$, c, $\rightarrow)$
q_1	c	$(q_1$, c, $\rightarrow)$

We can encode the states and the alphabet as:

q_0	q00
q_1	q01
h	q10
a	a00
b	a01
❑	a10
c	a11

We can then encode δ as:

```
(q00,a00,q01,a01,→),(q00,a01,q01,a00,→),(q00,a10,q10,a10,→),
(q00,a11,q00,a11,→),(q01,a00,q00,a11,→),(q01,a01,q00,a01,←),
(q01,a10,q00,a11,→),(q01,a11,q01,a11,→)
```

19 The Unsolvability of the Halting Problem

1) Consider the language L = {<M> : M accepts at least two strings}.
   a) Describe in clear English a Turing machine M that semidecides L.

   M generates the strings in Σ_M* in lexicographic order and uses dovetailing to interleave the computation of M on those strings. As soon as two computations accept, M halts and accepts.

   b) Suppose we changed the definition of L just a bit. We now consider:
   L' = {<M> : M accepts *exactly* 2 strings>.
   Can you tweak the Turing machine you described in part a to semidecide L'?

   No. M could discover that two strings are accepted. But it will never know that there aren't any more.

2) Consider the language L = {<M> : M accepts the binary encodings of the first three prime numbers}.
   a) Describe in clear English a Turing machine M that semidecides L.

   On input <M> do:
   1. Run M on `10`. If it rejects, loop.
   2. If it accepts, run M on `11`. If it rejects, loop.
   3. If it accepts, run M on `101`. If it accepts, accept. Else loop.

   This procedure will halt and accept iff M accepts the binary encodings of the first three prime numbers. If, on any of those inputs, M either fails to halt or halts and rejects, this procedure will fail to halt.

   b) Suppose (contrary to fact, as established by Theorem 19.2) that there were a Turing machine *Oracle* that decided H. Using it, describe in clear English a Turing machine M that decides L.

   On input <M> do:
   1. Invoke *Oracle*(<M, `10`>).
   2. If M would not accept, reject.
   3. Invoke *Oracle*(<M, `11`>).
   4. If M would not accept, reject.
   5. Invoke *Oracle*(<M, `101`>).
   6. If M would accept, accept. Else reject.

20 Decidable and Semidecidable Languages

1) Show that the set D (the decidable languages) is closed under:
   a) Union
   b) Concatenation
   c) Kleene star
   d) Reverse
   e) Intersection

All of these can be done by construction using deciding TMs. (Note that there's no way to do it with grammars, since the existence of an unrestricted grammar that generates some language L does not tell us anything about whether L is in D or not.)

a) Union is straightforward. Given a TM M_1 that decides L_1 and a TM M_2 that decides L_2, we build a TM M_3 to decide $L_1 \cup L_2$ as follows: Initially, let M_3 contain all the states and transitions of both M_1 and M_2. Create a new start state S and add transitions from it to the start states of M_1 and M_2 so that each of them begins in its start state with its read/write head positioned just to the left of the input. The accepting states of M_3 are all the accepting states of M_1 plus all the accepting states of M_2.

b) is a bit tricky. Here it is: If L_1 and L_2 are both in D, then there exist TMs M_1 and M_2 that decide them. From M_1 and M_2, we construct M_3 that decides L_1L_2. Since there is a TM that decides L_3, it is in D.

The tricky part is doing the construction. When we did this for FSMs, we could simply glue the accepting states of M_1 to the start state of M_2 with ε transitions. But that doesn't work now. Consider a machine that enters the state y when it scans off the edge of the input and finds a blank. If we're trying to build M_3 to accept L_1L_2, then there won't be a blank at the end of the first part. But we can't simply assume that that's how M_1 decides it's done. It could finish some other way.

So we need to build M_3 so that it works as follows: M_3 will use three tapes. Given some string w on tape 1, M_3 first nondeterministically guesses the location of the boundary between the first segment (a string from L_1) and the second segment (a string from L_2). It copies the first segment onto tape 2 and the second segment onto tape 3. It then simulates M_1 on tape 2. If M_1 accepts, it simulates M_2 on tape 3. If M_2 accepts, it accepts. If either M_1 or M_2 rejects, that path rejects.

There is a finite number of ways to carve the input string w into two segments. So there is a finite number of branches. Each branch must halt since M_1 and M_2 are deciding machines. So eventually all branches will halt. If at least one accepts, M_3 will accept. Otherwise it will reject.

2) Show that the set SD (the semidecidable languages) is closed under:
   a) Union
   b) Concatenation
   c) Kleene star
   d) Reverse
   e) Intersection

3) Let $L_1, L_2, \ldots, L_k$ be a collection of languages over some alphabet Σ such that:
   - For all $i \neq j$, $L_i \cap L_j = \varnothing$.
   - $L_1 \cup L_2 \cup \ldots \cup L_k = \Sigma^*$.
   - $\forall i$ (L_i is in SD).

   Prove that each of the languages L_1 through L_k is in D.

$\forall i$ ($\neg L_i = L_1 \cup L_2 \cup \ldots \cup L_{i-1} \cup L_{i+1} \cup \ldots \cup L_k$).
Each of these L_j's is in SD, so the union of all of them is in SD. Since L_i is in SD and so is its complement, it is in D.

4) If L_1 and L_3 are in D and $L_1 \subseteq L_2 \subseteq L_3$, what can we say about whether L_2 is in D?

L_2 may or may not be in D. Let L_1 be $\varnothing$ and let L_3 be Σ. Both of them are in D. Suppose L_2 is H. Then it is not in D. But now suppose that L_2 is {a}. Then it is in D.

5) Let L_1 and L_2 be any two decidable languages. State and prove your answer to each of the following questions:

a) Is it necessarily true that $L_1 - L_2$ is decidable?

Yes. The decidable languages are closed under complement and intersection, so they are closed under difference.

b) Is it possible that $L_1 \cup L_2$ is regular?

Yes. Every regular language is decidable. So Let L_1 and L_2 be {a}. $L_1 \cup L_2$ = {a}, and so is regular.

6) Let L_1 and L_2 be any two undecidable languages. State and prove your answer to each of the following questions:

a) Is it possible that $L_1 - L_2$ is regular?

Yes. Let $L_1 = L_2$. Then $L_1 - L_2 = \varnothing$, which is regular.

b) Is it possible that $L_1 \cup L_2$ is in D?

Yes. H $\cup$ ¬H = {<*M*, *w*>}.

7) Let M be a Turing machine that lexicographically enumerates the language L. Prove that there exists a Turing machine M' that decides L^R.

Since L is lexicographically enumerated by M, it is decidable. The decidable languages are closed under reverse. So L^R is decidable. Thus there is some Turing machine M' that decides it.

8) Construct a standard one-tape Turing machine M to enumerate the language:

{w : w is the binary encoding of a positive integer that is divisible by 3}.

Assume that M starts with its tape equal to ❑. Also assume the existence of the printing subroutine P, defined in Section 20.5.1. As an example of how to use P, consider the following machine, which enumerates L', where L' = {w : w is the unary encoding of an even number}:

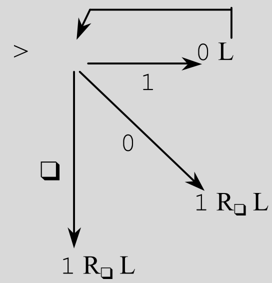
$> P\ R\ 1\ R\ 1$

You may find it useful to define other subroutines as well.

Define the subroutine *A* (Add1) as follows:

Input: ❑ $w_1\ w_2\ w_3 \ldots \underline{w}_n$ ❑ (encoding some integer k)

Output: ❑ $w_1\ w_2\ w_3 \ldots \underline{w}_m$ ❑ (encoding k+1)



The enumerating machine *M* is now:

> *R* 0 *A A A P*

9) Construct a standard one-tape Turing machine *M* to enumerate the language A^nB^n. Assume that *M* starts with its tape equal to ❑. Also assume the existence of the printing subroutine *P*, defined in Section 20.5.1.

>*P* L❑ a R❑ b

21 Decidability and Undecidability Proofs

1) For each of the following languages L, state whether it is in D, in SD/D, or not in SD. Prove your answer. Assume that any input of the form $<M>$ is a description of a Turing machine.

   a) {a}.

   D. L is finite and thus regular and context-free. By Theorem 20.1, every context-free language is in D.

   b) $<M> : \texttt{a} \in L(M)\}$.

   SD/D. Let R be a mapping reduction from H to L defined as follows:

   $R(<M, w>) =$

   1. Construct the description $<M\#>$ of a new Turing machine $M\#(x)$ that, on input x, operates as follows:
      1.1. Erase the tape.
      1.2. Write w on the tape.
      1.3. Run M on w.
      1.4. Accept.
   2. Return $<M\#>$.

   If *Oracle* exists, then $C = Oracle(R(<M, w>))$ decides L:

   - R can be implemented as a Turing machine.
   - C is correct: $M\#$ accepts everything or nothing, depending on whether M halts on w. So:
     - $<M, w> \in H$: M halts on w, so $M\#$ accepts all inputs, including a. *Oracle* accepts.
     - $<M, w> \notin H$: M does not halt on w, so $M\#$ accepts nothing. In particular, it does not accept a. *Oracle* rejects.

   But no machine to decide H can exist, so neither does *Oracle*.

   c) $\{<M> : L(M) = \{\texttt{a}\}\}$.

   ¬SD: Let R be a reduction from $\neg H = \{<M, w> : \text{TM } M \text{ does not halt on } w\}$ to L, defined as follows:

   $R(<M, w>) =$

   1. Construct the description of $M\#(x)$ that, on input x, operates as follows:
      1.1. If $x = \texttt{a}$, accept.
      1.2. Erase the tape.
      1.3. Write w.
      1.4. Run M on w.
      1.5. Accept.
   2. Return $<M\#>$.

   If *Oracle* exists and semidecides L, then $C = R(<M, w>))$ semidecides $\neg H$:

   - $<M, w> \in \neg H$: M does not halt on w, so $M\#$ accepts the string a and nothing else. So $L(M\#) = \{\texttt{a}\}$. *Oracle* accepts.
   - $<M, w> \notin \neg H$: M halts on w. $M\#$ accepts everything. So $L(M\#) \neq \{\texttt{a}\}$. *Oracle* does not accept.

   But no machine to semidecide $\neg H$ can exist, so neither does *Oracle*.

d) $\{<M_a, M_b> : \varepsilon \in L(M_a) - L(M_b)\}$.

$\neg$SD. Let R be a reduction from $\neg$H = {<M, w> : TM M does not halt on w} to L, defined as follows:
R(<M, w>) =
1. Construct the description of $M\#(x)$ that operates as follows:
   1.1. Erase the tape.
   1.2. Write w.
   1.3. Run M on w.
   1.4. Accept.
2. Construct the description of $M?(x)$ that, on input x, operates as follows:
   2.1. Accept.
3. Return <$M?$, $M\#$>.

If *Oracle* exists and semidecides L, then C = *Oracle*(R(<M, w>)) semidecides $\neg$H:

- R can be implemented as a Turing machine.
- C is correct: $M?$ accepts everything, including ε. $M\#$ accepts everything or nothing, depending on whether M halts on w. So:
  - <M, w> $\in \neg$H: M does not halt on w. $M\#$ gets stuck in step 1.3. $L(M\#) = \varnothing$. $L(M?) - L(M\#) = L(M?)$, which contains ε. So *Oracle* accepts.
  - <M, w> $\notin \neg$H: M halts on w. So $L(M\#) = \Sigma^*$. $L(M?) - L(M\#) = \varnothing$, which does not contain ε. So *Oracle* does not accept.

But no machine to semidecide $\neg$H can exist, so neither does *Oracle*.

e) $\{<M_a, M_b> : L(M_a) = L(M_b) - \{\varepsilon\}\}$.

$\neg$SD.

f) $\{<M_a, M_b> : L(M_a) \neq L(M_b)\}$.

$\neg$SD. Let R be a reduction from $\neg$H = {<M, w> : TM M does not halt on w} to L, defined as follows:
R(<M, w>) =
1. Construct the description of $M\#(x)$ that, on input x, operates as follows:
   1.1. Erase the tape.
   1.2. Write w.
   1.3. Run M on w.
   1.4. Accept.
2. Construct the description of $M?(x)$ that, on input x, operates as follows:
   2.1. Accept.
3. Return <$M?$, $M\#$>.

If *Oracle* exists and semidecides L, then C = *Oracle*(R(<M, w>)) semidecides $\neg$H:

- R can be implemented as a Turing machine.
- C is correct: $L(M?) = \Sigma^*$. $M\#$ accepts everything or nothing, depending on whether M halts on w. So:
  - <M, w> $\in \neg$H: M does not halt on w. $M\#$ gets stuck in step 1.3. $L(M\#) = \varnothing$. $L(M?) \neq L(M\#)$. So *Oracle* accepts.
  - <M, w> $\notin \neg$H: M halts on w. So $L(M\#) = \Sigma^*$. $L(M?) = L(M\#)$. So *Oracle* does not accept.

But no machine to semidecide $\neg$H can exist, so neither does *Oracle*.

g) {<M, w> : M, when operating on input w, never moves to the right on two consecutive moves}.

D. Notice that $M = (K, \Sigma, \Gamma, \delta, s, H)$ must move either to the right or the left on each move. If it cannot move right on two consecutive moves, then every time it moves right, it must next move back left. So it will never be able to read more than the first square of its input tape. It can, however, move left indefinitely. That part of the tape is already known to contain only blanks. M can write on the tape as it moves left, but it cannot ever come back to read anything that it has written except the character it just wrote and the one immediately to its right. So the rest of the tape is no longer an effective part of *M*'s configuration. We need only consider the current square and one square on either side of it. Thus the number of effectively distinct configurations of *M* is $max = |K| \cdot |\Gamma|^3$. Once *M* has executed *max* steps, it must either halt or be in a loop. If the latter, it will just keep doing the same thing forever. So the following procedure decides *L*:

Run *M* on *w* for $|K| \cdot |\Gamma|^3 +1$ moves or until *M* halts or moves right on two consecutive moves:

- If *M* ever moves right on two consecutive moves, halt and reject.
- If *M* halts without doing that or if it has not done that after $|K| \cdot |\Gamma|^3 +1$ moves, halt and accept.

h) {<*M*> : *M* is the only Turing machine that accepts *L*(*M*)}.

D. *L* = ∅, since any language that is accepted by some Turing machine is accepted by an infinite number of Turing machines.

i) {<*M*> : *L*(*M*) contains at least two strings}.

SD/D: The following algorithm semidecides *L*:

Run *M* on the strings in Σ* in lexicographic order, interleaving the computations. As soon as two such computations have accepted, halt.

Proof not in D: *R* is a reduction from H = {<*M*, *w*> : TM M halts on *w*} to *L*, defined as follows:

R(<*M*, *w*>) =

1. Construct the description of *M*#(*x*) that, on input *x*, operates as follows:
    1.1. Erase the tape.
    1.2. Write *w* on the tape.
    1.3. Run *M*.
    1.4. Accept.
2. Return <*M*#>.

If *Oracle* exists and decides *L*, then *C* = *Oracle*(*R*(<*M*, *w*>)) decides H:

- <*M*, *w*> ∈ H: *M* halts on *w* so *M#* accepts everything and thus accepts at least two strings, so *Oracle* accepts.
- <*M*, *w*> ∉ H: *M* doesn't halt on *w* so *M#* doesn't halt and thus accepts nothing and so does not accept at least two strings so *Oracle* rejects.

But no machine to decide H can exist, so neither does *Oracle*.

j) {<*M*> : *M* rejects at least two even length strings}.

SD/D: The following algorithm semidecides *L*:

Run *M* on the even length strings in Σ* in lexicographic order, interleaving the computations. As soon as two such computations have rejected, halt.

Proof not in D: *R* is a reduction from H = {<*M*, *w*> : TM *M* halts on *w*} to *L*, defined as follows:

R(<*M*, *w*>) =

1. Construct the description of *M*#(*x*) that, on input *x*, operates as follows:
    1.1. Erase the tape.
    1.2. Write *w* on the tape.
    1.3. Run *M*.
    1.4. Reject.
2. Return <*M*#>.

If *Oracle* exists and decides *L*, then *C* = *Oracle*(*R*(<*M*, *w*>)) decides H:

- <*M*, *w*> ∈ H: *M* halts on *w* so *M#* rejects everything and thus rejects at least two even length strings, so *Oracle* accepts.
- <*M*, *w*> ∉ H: *M* doesn't halt on *w* so *M#* doesn't halt and thus rejects nothing and so does not reject at least even length two strings. *Oracle* rejects.

But no machine to decide H can exist, so neither does *Oracle*.

k) {<*M*> : *M* halts on all palindromes}.

¬SD: Assume that Σ ≠ ∅. *R* is a reduction from ¬H = {<*M*, *w*> : TM *M* does not halt on *w*} to *L*, defined as follows:

R(<*M*, *w*>) =
1. Construct the description of *M#*(*x*) that, on input *x*, operates as follows:
   1.1. Save its input *x* on a second tape.
   1.2. Erase the tape.
   1.3. Write *w*.
   1.4. Run *M* on *w* for |*x*| steps or until it halts.
   1.5. If *M* would have halted, then loop.
   1.6. Else halt.
2. Return <*M#*>.

If *Oracle* exists and semidecides *L*, then *C* = *R*(<*M*, *w*>)) semidecides ¬H:

- <*M*, *w*> ∈ ¬H: *M* does not halt on *w*, so *M#* always gets to step 1.6. So it halts on everything, including all palindromes, so *Oracle* accepts.
- <*M*, *w*> ∉ ¬H: *M* halts on *w*. Suppose it does so in *k* steps. Then, for all strings of length *k* or more, *M#* loops at step 1.5. For any *k*, there is a palindrome of length greater than *k*. So *M#* fails to accept all palindromes. So *Oracle* does not accept.

But no machine to semidecide ¬H can exist, so neither does *Oracle*.

l) {<*M*> : *L*(*M*) is context-free}.

¬SD. *R* is a reduction from ¬H = {<*M*, *w*> : TM *M* does not halt on *w*} to *L*, defined as follows:

R(<*M*, *w*>) =
1. Construct the description of *M#*(*x*) that, on input *x*, operates as follows:
   1.1. Save *x*.
   1.2. Erase the tape.
   1.3. Write *w*.
   1.4. Run *M* on *w*.
   1.5. If *x* ∈ $A^nB^nC^n$, accept. Else loop.
2. Return <*M#*>.

If *Oracle* exists and semidecides *L*, then *C* = *Oracle*(*R*(<*M*, *w*>)) semidecides ¬H:

- <*M*, *w*> ∈ ¬H: *M* does not halt on *w*. *M#* gets stuck in step 1.4. So *L*(*M#*) = ∅, which is context-free. So *Oracle* accepts.
- <*M*, *w*> ∉ ¬H: *M* halts on *w*. So *L*(*M#*) = $A^nB^nC^n$, which is not context-free. So *Oracle* does not accept.

But no machine to semidecide ¬H can exist, so neither does *Oracle*.

m) {<*M*> : *L*(*M*) is not context-free}.

¬SD. *R* is a reduction from ¬H = {<*M*, *w*> : TM *M* does not halt on *w*} to *L*, defined as follows:

R(<*M*, *w*>) =

1. Construct the description of *M#*(*x*) that, on input *x*, operates as follows:
   1.1. If $x \in A^nB^nC^n$, accept.
   1.2. Erase the tape.
   1.3. Write *w*.
   1.4. Run *M* on *w*.
   1.5. Accept.
2. Return <*M#*>.

If *Oracle* exists and semidecides *L*, then *C* = *Oracle*(*R*(<*M*, *w*>)) semidecides ¬H:

- <*M*, *w*> ∈ ¬H: *M* does not halt on *w*. *M#* gets stuck in step 1.4. So $L(M^*) = A^nB^nC^n$, which is not context-free. So *Oracle* accepts.
- <*M*, *w*> ∉ ¬H: *M* halts on *w*. So *L*(*M#*) = Σ*, which is context -free. So *Oracle* does not accept.

But no machine to semidecide ¬H can exist, so neither does *Oracle*.

n) {<*M*> : $A_\#(L(M)) > 0$}, where $A_\#(L) = |L \cap \texttt{a}^*|$.

SD/D: The following algorithm semidecides *L*: Lexicographically enumerate the strings in `a`* and run them through *M* in dovetailed mode. If *M* ever accepts a string, accept.

We show not in D by reduction: *R* is a reduction from H to *L*, defined as follows:

R(<*M*, *w*>) =

1. Construct the description of *M#*(*x*) that, on input *x*, operates as follows:
   1.1 Erase the tape.
   1.2 Write *w*.
   1.3 Run *M* on *w*.
   1.4 Accept.
2. Return <*M#*>.

If *Oracle* exists and decides *L*, then *C* = *Oracle*(*R*(<*M*, *w*>)) decides H:

- <*M*, *w*> ∈ H: *M* halts on *w*. *M#* accepts everything, including strings in `a`*. So *Oracle* accepts.
- <*M*, *w*> ∉ ¬H: *M* does not halt on *w*. *M#* accepts nothing. So *Oracle* rejects.

But no machine to decide H can exist, so neither does *Oracle*.

o) {<*M*> : |*L*(*M*)| is a prime integer greater than 0}.

¬SD: Assume that Σ ≠ ∅. *R* is a reduction from ¬H = {<*M*, *w*> : TM *M* does not halt on *w*} to *L*, defined as follows:

R(<*M*, *w*>) =

1. Construct the description of *M#*(*x*) that, on input *x*, operates as follows:
   1.1. If *x* is one of the first two strings (lexicographically) in Σ*, accept.
   1.2. Erase the tape.
   1.3. Write *w* on the tape.
   1.4. Run *M* on *w*.
   1.5. Accept.
2. Return <*M#*>.

If *Oracle* exists and semidecides *L*, then *C* = *R*(<*M*, *w*>)) semidecides ¬H:

- <*M*, *w*> ∈ ¬H: *M* does not halt on *w*, so the *M#* accepts only the two strings that it accepts in step 1.1. So |*L*(*M#*)| = 2, which is greater than 0 and prime, so *Oracle* accepts.
- <*M*, *w*> ∉ ¬H: *M* halts on *w* so, *M#* accepts everything else at step 1.5. There is an infinite number of strings over any nonempty alphabet, so *L*(*M#*) is infinite. Its cardinality is not a prime integer. So *Oracle* does not accept.

But no machine to semidecide ¬H can exist, so neither does *Oracle*.

p) {<*M*> : there exists a string *w* such that |*w*| < |<*M*>| and that *M* accepts *w*}.

SD/D: The following algorithm semidecides *L*:

> Run *M* on the strings in Σ* of length less than |<*M*>|, in lexicographic order, interleaving the computations. If any such computation halts, halt and accept.

Proof not in D: *R* is a reduction from H = {<*M*, *w*> : TM *M* halts on *w*} to *L*, defined as follows:

R(<*M*, *w*>) =

1. Construct the description <*M#*> of a new Turing machine *M#*(*x*) that, on input *x*, operates as follows:
   1.1. Erase the tape.
   1.2. Write *w* on the tape.
   1.3. Run *M* on *w*.
   1.4. Accept.
2. Return <*M#*>.

If *Oracle* exists and decides *L*, then *C* = *Oracle*(*R*(<*M*, *w*>)) decides H:

- <*M*, *w*> ∈ H: *M* halts on *w* so *M#* accepts everything. So, in particular, it accepts ε, which is a string of length less than |<*M*>|, so *Oracle*(<*M#*>) accepts.
- <*M*, *w*> ∉ H: *M* doesn't halt on *w* so *M#* doesn't halt and thus accepts nothing. So, in particular there is no string of length less than |<*M*>| that *M#* accepts, so *Oracle*(<*M#*>) rejects.

But no machine to decide H can exist, so neither does *Oracle*.

q) {<*M*> : *M* does not accept any string that ends with `0`}.

¬SD: *R* is a reduction from ¬H = {<*M*, *w*> : TM *M* does not halt on *w*} to *L*, defined as follows:

R(<*M*, *w*>) =

1. Construct the description of *M#*(*x*) that, on input *x*, operates as follows:
   1.1. Erase the tape.
   1.2. Write *w* on the tape.
   1.3. Run *M* on *w*.
   1.4. Accept.
2. Return <*M#*>.

If *Oracle* exists and semidecides *L*, then *C* = *R*(<*M*, *w*>)) semidecides ¬H:

- <*M*, *w*> ∈ ¬H: *M* does not halt on *w* so *M#* accepts nothing and so, in particular, accepts no string that ends in `0`. So *Oracle* accepts.
- <*M*, *w*> ∉ ¬H: *M* halts on *w* so *M#* accepts everything, including all strings that end in `0`. Since *M#* does accept strings that end in `0`, *M#* is not in *L* and *Oracle* does not accept.

But no machine to semidecide ¬H can exist, so neither does *Oracle*.

r) {<*M*> : there are at least two strings *w* and *x* such that *M* halts on both *w* and *x* within some number of steps *s*, and $s < 1000$ and *s* is prime}.

D. Note that in any fixed number *s* steps, *M* can examine no more than *s* squares of its tape. So if it is going to accept any string *w* that is longer than *s*, it must also accept a string *w'* that is no longer than *s* and that is an initial substring of *w*. So the following algorithm decides *L*:

Run *M* on all strings in Σ* of length between 0 and 1000. Try each for 1000 steps or until the computation halts:

- If at least two such computations halted in some prime number of steps *s*, accept.
- Else reject.

s) {<*M*> : there exists an input on which TM *M* halts in fewer than |<*M*>| steps}.

D. In |<*M*>| steps, *M* can examine no more than |<*M*>| squares of its tape. So the following algorithm decides *L*:

Run *M* on all strings in Σ* of length between 0 and |<*M*>|. Try each for |<*M*>| steps or until the computation halts:

- If at least one such computation halted, accept.
- Else reject.

It isn't necessary to try any longer strings because, if *M* accepts some longer string, it does so by looking at no more than |<*M*>| initial characters. So it would also accept the string that contains just those initial characters. And we'd have discovered that.

t) {<*M*> : *L*(*M*) is infinite}.

¬SD. Assume that Σ ≠ ∅. R is a reduction from ¬H = {<*M*, *w*> : TM M does not halt on *w*} to *L*, defined as follows:

R(<*M*, *w*>) =

1. Construct the description of *M*#(*x*) that, on input *x*, operates as follows:
   1.1. Save its input *x* on a second tape.
   1.2. Erase the tape.
   1.3. Write *w*.
   1.4. Run *M* on *w* for |*x*| steps or until it halts.
   1.5. If *M* would have halted, then loop.
   1.6. Else accept.
2. Return *Oracle*(<*M*#>)

If *Oracle* exists and semidecides *L*, then *R* semidecides ¬H:

- <*M*, *w*> ∈ ¬H: *M* does not halt on *w*. So *M*# always makes it to step 1.6. It accepts everything, which is an infinte set. So *Oracle* accepts.
- <*M*, *w*> ∉ ¬H: *M* halts on *w*. Suppose it does so in *k* steps. Then *M*# loops on all strings of length *k* or greater. It accepts strings of length less than *k*. But that set is finite. So *Oracle* does not accept.

But no machine to semidecide ¬H can exist, so neither does *Oracle*.

u) {<*M*> : *L*(*M*) is uncountably infinite}.

D. *L* = ∅, since every Turing machine *M* = (*K*, Σ, Γ, δ, *s*, *H*) accepts some subset of Σ* and |Σ*| is countably infinite. So *L* is not only in D, it is regular.

7) Show that each of the following questions is undecidable by recasting it as a language recognition problem and showing that the corresponding language is not in D:

a) Given a program P, input x, and a variable n, does P, when running on x, ever assign a value to n?

L = {<P, x, n> : P, when running on x, ever assigns a value to n}. We show that L is not in D by reduction from H. Define:

R(<M, w>) =

1. Construct the description <P> of a program P that ignores its input and operates as follows:
   1.1. Erase a simulated tape.
   1.2. Write w on the tape.
   1.3. Simulate running M on w.
   1.4. Set n to 0.
2. Return <P, ε, n>.

If *Oracle* exists and decides L, then C = *Oracle*(R(<M, w>)) decides H. R can be implemented as a Turing machine. And C is correct:

- <M, w> ∈ H: M halts on w, so P, regardless of its input, assigns a value to n. *Oracle*(<P, ε, n>) accepts.
- <M, w> ∉ H: M does not halt on w, so P, regardless of its input, fails to assign a value to n. . *Oracle*(<P, ε, n>) rejects.

But no machine to decide H can exist, so neither does *Oracle*.

b) Given a program P and code segment S in P, does P reach S on every input (in other words, can we guarantee that S happens)?

c) Given a program P and a variable x, is x always initialized before it is used?

d) Given a program P and a file f, does P always close f before it exits?

e) Given a program P with an array reference of the form $a[i]$, will i, at the time of the reference, always be within the bounds declared for the array?

f) Given a program P and a database of objects d, does P perform the function f on all elements of d?

L = {<P, d, f> : P performs f on every element of d}. We show that L is not in D by reduction from H. Define:

R(<M, w>) =

1. Create a database D with one record r.
2. Create the function f that writes the value of the first field of the database object it is given.
3. Construct the description <P> of a program P that ignores its input and operates as follows:
   3.1. Erase a simulated tape.
   3.2. Write w on the tape.
   3.3. Simulate running M on w.
   3.4. Run f on r.
4. Return <P, D, f>.

If *Oracle* exists and decides L, then $C = Oracle(R(<M, w>))$ decides H. R can be implemented as a Turing machine. And C is correct:

- $<M, w> \in$ H: M halts on w, so P, regardless of its input, runs f on r. $Oracle(<P, D, f>)$ accepts.
- $<M, w> \notin$ H: M does not halt on w, so P, regardless of its input, fails to run f on r. . $Oracle(<P, D, f>)$ rejects.

But no machine to decide H can exist, so neither does *Oracle*.

10) Do the other half of the proof of Rice's Theorem, i.e., show that the theorem holds if $P(\varnothing) = True$.

The easiest way to do this is to use a reduction that is not a mapping reduction. We simply invert the reduction that we did to prove the first half. So we proceed as follows. Assume that $P(\varnothing) = True$. Since P is nontrivial, there is some SD language L_F such that $P(L_F)$ is *False*. Since L_F is in SD, there exists some Turing machine K that semidecides it.

Define:

$R(<M, w>) =$

1. Construct the description $<M\#>$ of a new Turing machine $M\#(x)$ that, on input x, operates as follows:
   1.1. Copy its input x to a second tape.
   1.2. Erase the tape.
   1.3. Write w on the tape.
   1.4. Run M on w.
   1.5. Put x back on the first tape and run K on x.
2. Return $<M\#>$.

$\{R, \neg\}$ is a reduction from H to L_2. If *Oracle* exists and decides L, then $C = \neg Oracle(R(<M, w>))$ decides H. R can be implemented as a Turing machine. And C is correct:

- If $<M, w> \in$ H: M halts on w, so $M\#$ makes it to step 1.5. So $M\#$ does whatever K would do. So $L(M\#) = L(K)$ and $P(L(M\#)) = P(L(K))$. We chose K precisely to assure that $P(L(K))$ is *False*, so $P(L(M\#))$ must also be *False*. *Oracle* decides P. $Oracle(<M\#>)$ rejects so C accepts.
- If $<M, w> \notin$ H: M does not halt on w. $M\#$ gets stuck in step 1.4 and so accepts nothing. $L(M\#) = \varnothing$. By assumption, $P(\varnothing) = True$. *Oracle* decides P. $Oracle(<M\#>)$ accepts so C rejects.

But no machine to decide H can exist, so neither does *Oracle*.

11) For each of the following languages L, do two things:
   i) State whether or not Rice's Theorem has anything to tell us about the decidability of L.
   ii) State whether L is in D, SD/D, or not in SD.

a) $\{<M> : M$ accepts all strings that start with `a`$\}$.

Rice's Theorem applies and tells us that L is not in D. It is also true that L is not in SD.

b) $\{<M> : M$ halts on ε in no more than 1000 steps$\}$.

Rice's Theorem does not apply. L is in D. It can be decided by simply running M on ε for 1000 steps or until it halts.

c) $\neg L_1$, where $L_1 = \{<M> : M$ halts on all strings in no more than 1000 steps$\}$.

Rice's Theorem does not apply. L_1 is in D. The key to defining a decision procedure for it is the observation that if M is allowed to run for only 1000 steps, it must make its decision about whether to accept an input string w after looking at no more than the first 1000 characters of w. So we can decide L_1

by doing the following: Lexicographically enumerate the strings of length up 1000 drawn from the alphabet of M. For each, run M for 1000 steps or until it halts. If M halted on all of them, then it must also halt on all longer strings. So accept. Otherwise, reject. Since the decidable languages are closed under complement, $\neg L_1$ must be in D if L_1.

d) $\{<M, w> : M \text{ rejects } w\}$.

Rice's Theorem does not apply. Note that the definition of this language does not ask about the language that M accepts. Failure to reject could mean either that M accepts or that it loops. L is in SD.

12) Use Rice's Theorem to prove that each of the following languages is not in D:

a) $\{<M> : \text{Turing machine } M \text{ accepts at least two odd length strings}\}$.

We define P as follows:

- Let P be defined on the set of languages accepted by some Turing machine M. Let it be *True* if $L(M)$ contains at least two odd length strings and *False* otherwise.
- The domain of P is the SD languages since it is those languages that are accepted by some Turing machine M.
- P is nontrivial since $P(\{\texttt{a}, \texttt{aaa}\})$ is *True* and $P(\varnothing)$ is *False*.

Thus $\{<M> : \text{Turing machine } M \text{ accepts at least two odd length strings}\}$ is not in D.

b) $\{<M> : M \text{ is a Turing machine and } |L(M)| = 12\}$.

We define P as follows:

- Let P be defined on the set of languages accepted by some Turing machine M. Let it be *True* if $|L(M)|$ is 12 and *False* otherwise.
- The domain of P is the SD languages since it is those languages that are accepted by some Turing machine M.
- P is nontrivial since $P(\{\texttt{a}, \texttt{aa}, \texttt{aaa}, \texttt{aaaa}, \texttt{aaaaa}, \texttt{aaaaaa}, \texttt{b}, \texttt{bb}, \texttt{bbb}, \texttt{bbbb}, \texttt{bbbbb}, \texttt{bbbbbb}\})$ is *True* and $P(\varnothing)$ is *False*.

Thus $\{<M> : M \text{ is a Turing machine and } |L(M)| = 12\}$ is not in D.

20) If L_1 and L_2 are decidable languages and $L_1 \subseteq L \subseteq L_2$, must L be decidable? Prove your answer.

No. Let $L_1 = \varnothing$. Let $L_2 = \{<M>\}$. Let $L = \{<M> : M \text{ accepts } \varepsilon\}$, which is not decidable.