Chapter 4

Decidability and Undecidability

4.1 What is Computability?

From an informal or intuitive perspective what might we mean by computability? One natural interpretation is that something is computable if it can be calculated by a systematic procedure; we might think of this as a process that can be described by one person and carried out by another.

We will now assert that any such process can be written as a computer program. This is known as the Church-Turing thesis because they were the first to give formal mathematical specifications of what constituted an algorithm in form of the lambda calculus and Turing Machines, respectively. Note that the claim is a thesis; it is inherently not something provable, for the notion of a systematic process is unavoidably imprecise.

We remark that an instruction such as “guess the correct answer” does not seem to be systematic. An instruction such as “try all possible answers” is less clear cut: it depends on whether the possible answers are finite or infinite in number. For specificity, henceforth we will imagine working with a specific programming language, and using this as the definition of reasonable instructions (you can think of this as Java, C, or whatever is your preferred programming language). Actually, in practice we will describe algorithms in pseudo-code and English, but with enough detail that their programmability should be clear.

We define computability in terms of language recognition.

**Definition 4.1.1.** $L \subseteq \Sigma^*$ is computable if there is a program $P$ with two possible outputs, “Recognize” and “Reject” and on input $x \in \Sigma^*$, $P$ outputs “Recognize” if $x \in L$ and “Reject” if $x \notin L$. We will also say that $P$ decides $L$ (in the sense of decides set membership).

**Remark.** Often, one wants to compute some function $f(x)$ of the input $x$. By allowing 2-input programs we can define computable functions $f$ as follows: $f$ is computable if there is a 2-input program $P$, such that on input $(x, y)$, $P$ outputs “Recognize” if $f(x) = y$ and “Reject” if $f(x) \neq y$. 
4.2 Encodings

Often, we will want to treat the text of one program as a possible input to another program. To this end, we take the conventional approach that the program symbols are encoded in ASCII or more simply in binary. We do the same for the input alphabet. Of course, if we are not trying to treat a program as a possible input to another program, we can have distinct alphabets for programs and inputs.

As it turns out, not all languages are computable (indeed, we have already seen that the halting problem is not computable). First, however, we are going to see some computable languages.

The first languages we are going to look at capture properties of automata. To this end, we need an agreed method for writing the description of an automata. This is similar to a standard input format for a graph, for example. To describe a DFA we specify it as a sequence of 4 sets: \( \Sigma \), the input alphabet, \( V \), the set of vertices, \( \text{start} \in V \), the start vertex, \( F \subseteq V \), the set of recognizing vertices, and \( \delta \), a description of the edges and their labels. If \( \Sigma = \{a_1, a_2, \ldots, a_k\}, V = \{v_1, \ldots, v_r\} \) we could write this as a string:

\[
(\{a_1, a_2, \ldots, a_k\}, \{v_1, v_2, \ldots, v_r\}, \text{start}, \{v_{i_1}, v_{i_2}, \ldots \},
\{(v_1, a_1) \rightarrow v_{j_1}, (v_1, a_2) \rightarrow v_{j_2}, \ldots, (v_1, a_k) \rightarrow v_{j_k},
(v_2, a_2) \rightarrow v_{j_{k+1}}, \ldots, (v_r, a_k) \rightarrow v_{j_{kr}}\}),
\]

where \( F = \{v_{i_1}, v_{i_2}, \ldots, v_{i_l}\} \) and \( \delta(v_h, a_i) = v_{j_{(h-1)k+i}} \) for \( 1 \leq h \leq r, 1 \leq i \leq k \).

There is a further difficulty that we would like to use one alphabet to describe many different machines. One way of doing this is to write the vertices and the alphabet characters in binary. We then see that the above DFA description uses just 8 distinct characters: \( \{, , (, , )\), 0, 1, \( \rightarrow\), . We could reduce this to two characters, 0 and 1 say, by encoding each of these characters using three 0s and 1s.

We will view an input string as a descriptor of a DFA only if it has the above format, and in addition:

- \( a_i \neq a_j \) for \( i \neq j \), i.e. the relevant binary strings are not equal.
- \( v_i \neq v_j \) for \( i \neq j \).
- \( \text{start} = v_i \) for some \( i, 1 \leq i \leq r \).
- For each \( h, 1 \leq h \leq j, v_{i_h} = v_l \), for some \( l, 1 \leq l \leq r \).
- In each expression \( (v_h, a_i) \rightarrow v_{j_{(h-1)k+i}}, v_{j_{(h-1)k+i}} = v_l \) for some \( l, 1 \leq l \leq r \).

As this is rather tedious, henceforth we leave the description of correct encodings to the reader (to imagine). We will simply use angle brackets to indicate a suitable encoding in a convenient alphabet (binary say). So \( \langle M \rangle \) denotes an encoding of machine \( M \), \( \langle w \rangle \) an
4.3. DECIDABILITY OF REGULAR LANGUAGE PROPERTIES

We are now ready to look at the decidability of some languages.

4.3 Decidability of Regular Language Properties

**Example 4.3.1.** Rec-DFA = \{⟨M, w⟩ | M is a DFA and M recognizes input w\}.

**Claim 4.3.2.** Rec-DFA is decidable.

**Proof.** To prove the claim we simply need to give an algorithm \(A\) that on input ⟨M, w⟩ determines whether M recognizes its input. Such an algorithm is easily seen at a high level: \(A\) first checks whether the input is legitimate and if not it rejects. By legitimate we mean that ⟨M, w⟩ is the encoding of a DFA, followed by the encoding of a string w over the input alphabet for M. If the input is legitimate \(A\) continues by simulating M on input w: \(A\) keeps track of the the current destination vertex as M reads its input w. When (in \(A\)’s simulation) M has read all of w, \(A\) checks whether the destination vertex M has reached is a final vertex and outputs accordingly: \(A\) outputs “Recognize” if M’s destination on input w is a final vertex, and \(A\) outputs “Reject” otherwise.

\[
A(⟨M, x⟩) = \begin{cases} 
  \text{“Recognize”} & \text{if } M \text{ recognizes } x \\
  \text{“Reject”} & \text{if } M \text{ does not recognize } x 
\end{cases}
\]

The details are a bit more painstaking: given the current vertex reached by M and the next character in w, \(A\) looks up the vertex reached by scanning the edge descriptions (the triples \((p, a) \rightarrow q\)). In yet more detail, \(A\) stores the current vertex in a variable, the input w in a linked list, a pointer to the next character of w to be read, and the DFA in adjacency list format.

The details of how to implement algorithm \(A\) should be clear at this point. As they are not illuminating we are not going to spell them out further.

**Note.** Henceforth, a description at the level of detail of the first paragraph of the above proof will suffice. Further, we will take it as given that there is an initial step in our algorithms to check that the inputs are legitimate.

Let \(P_{\text{RecDFA}}\) be the program implementing the just described algorithm \(A\) deciding the language of Example 4.3.1. So \(P_{\text{RecDFA}}\) takes inputs ⟨M, x⟩ (strictly, ⟨M, x⟩) and outputs “Recognize” if M recognizes x and outputs “Reject” if M does not recognize x. This is a notation we will use repeatedly. If Prop is a decidable property (i.e. the language \(L = \{w \mid \text{Prop}(w) \text{ is true}\} \) is decidable) then \(P_{\text{Prop}}\) will be a program that decides \(L\); we will also say that \(P_{\text{Prop}}\) decides Prop.
Example 4.3.3. Rec-NFA = \{⟨M, w⟩ | M is a description of an NFA and M decides w\}.

Claim 4.3.4. Rec-NFA is decidable.

Proof. The following algorithm \(A_{\text{Rec-NFA}}\) decides Rec-NFA. Given input \(⟨M, w⟩\), it simulates \(M\) on input \(w\) by keeping track of all possible destination vertices on reading the first \(i\) characters of \(w\), for \(i = 0, 1, 2, \ldots\) in turn. \(M\) recognizes \(w\) exactly if one of its destinations on reading all of \(w\) is a final vertex, and consequently \(A_{\text{Rec-NFA}}\) outputs “Recognize” if it finds \(M\) on input \(w\) has a destination vertex which is a final vertex and outputs “Reject” otherwise.

Example 4.3.5. Rec-RegExp = \{⟨r, w⟩ | r is a regular expression that generates w\}.

Claim 4.3.6. Rec-RegExp is decidable.

Proof. The following algorithm \(A_{\text{Rec-RegExp}}\) decides Rec-RegExp. Given input \(⟨r, w⟩\), it begins by building an NFA \(M_r\) recognizing the language \(L(r)\) described by \(r\), using the procedure from Chapter 2, Lemma 2.4.1. \(A_{\text{Rec-RegExp}}\) then forms the encoding \(⟨M_r, w⟩\) and simulates the program \(P_{\text{Rec-NFA}}\) from Example 4.3.3 on input \(⟨M_r, w⟩\). \(A_{\text{Rec-RegExp}}\’s output (“Recognize” or “Reject”) is the same as the one given by \(P_{\text{Rec-NFA}}\) on input \(⟨M_r, w⟩\).

This is correct for \(M_r\) recognizes \(w\) if and only if \(w\) is one of the strings described by \(r\).

This procedure is taking advantage of an already constructed program and using it as a subroutine. This is a powerful tool which we are going to be using repeatedly.

Example 4.3.7. Empty-DFA = \{⟨M⟩ | M is a DFA and \(L(M) = \emptyset\)\}.

Note that \(L(M) = \emptyset\) exactly if no final vertex of \(M\) is reachable from its start vertex. It is easy to give a graph search algorithm to test this.

Claim 4.3.8. Empty-DFA is decidable.

Proof. The following algorithm \(A_{\text{Empty-DFA}}\) decides Empty-DFA. Given input \(⟨M⟩\), it determines the collection of vertices reachable from \(M\’s start vertex. If this collection includes a final vertex then the algorithm outputs “Reject” and otherwise it outputs “Recognize”.

Example 4.3.9. Equal-DFA = \{⟨MA, MB⟩ | MA and MB are DFAs and \(L(M_A) = L(M_B)\)\}.

Claim 4.3.10. Equal-DFA is decidable.

Proof. Let \(A = L(M_A)\) and \(B = L(M_B)\). We begin by observing that there is a DFA \(\widetilde{M}_{AB}\) such that \(L(\widetilde{M}_{AB}) = \emptyset\) exactly if \(A = B\). For let \(C = (A \cap \overline{B}) \cup (\overline{A} \cap B)\). Clearly, if \(A = B\), \(C = \emptyset\). While if \(C = \emptyset\), \(A \cap \overline{B} = \emptyset\), so \(A \subseteq \overline{B} = B\); similarly \(\overline{A} \cap B = \emptyset\), so \(B \subseteq A\); together, these imply \(A = B\).

But given DFAs \(M_A\) and \(M_B\), we can construct DFAs \(\overline{M}_A\) and \(\overline{M}_B\) to recognize \(\overline{A}\) and \(\overline{B}\) respectively. Then using \(M_A\) and \(\overline{M}_B\) we can construct DFA \(M_{AB}\) to recognize
4.3. DECIDABILITY OF REGULAR LANGUAGE PROPERTIES

A ∩ B and $M_{AB}$ to recognize $\overline{A} \cap B$. Given $M_{AB}$, $M_{AB}$ we can construct $\widetilde{M}_{AB}$ to recognize $(A \cap B) \cup (\overline{A} \cap B)$.

So the algorithm $A_{\text{Equal-DFA}}$ to decide Equal-DFA, given input $\langle M_A, M_B \rangle$, constructs $\widetilde{M}_{AB}$ and forms the encoding $\langle \widetilde{M}_{AB} \rangle$. $A_{\text{Equal-DFA}}$ then simulates program $P_{\text{Empty-DFA}}$ from the preceding example on input $\langle \widetilde{M}_{AB} \rangle$. $A_{\text{Equal-DFA}}$ outputs the result of the simulation of $P_{\text{Empty-DFA}}$ on input $\langle \widetilde{M}_{AB} \rangle$.

This is correct for $P_{\text{Empty-DFA}}$ outputs “Recognize” exactly if $L(\widetilde{M}_{AB}) = \phi$ which is the case exactly if $A = B$.

We have now given two examples of a use of a subroutine in a very particular form. More specifically, program $P$ has used program $Q$ as a subroutine and then used the answer of $Q$ to compute its own output. In these two examples, the calculation of $P$’s output has been the simplest possible: the output of $Q$ has become the output of $P$.

We call this form of algorithm design a reduction. If we have a program (or algorithm) $Q$ that decides a language $A$ (e.g. Empty-DFA), and we give a program to decide language $B$ (e.g. Equal-DFA) using $Q$ as a subroutine then we say we have reduced language $B$ to language $A$. What this means is that if we know how to decide language $A$ we have now also demonstrated how to decide language $B$.

**Example 4.3.11.** Inf-DFA = $\{\langle M \rangle \mid M$ is a DFA and $L(M)$ is infinite$\}$.

**Claim 4.3.12.** Inf-DFA is decidable.

**Proof.** Note that $L(M)$ is infinite exactly if there is a path which includes a cycle and which goes from $M$’s start vertex to some final vertex. This property is readily tested by the following algorithm $A_{\text{Inf-DFA}}$.

**Step 1.** $A_{\text{Inf-DFA}}$ identifies the non-trivial strong components\(^1\) of $M$’s graph, that is those that contain at least one edge (so any vertices with a self-loop will be in a non-trivial strong component).

**Step 2.** $A_{\text{Inf-DFA}}$ forms the reduced graph, in which every strong component is replaced by a single vertex, and in addition it marks each non-trivial strong component (or rather the corresponding vertices). Further, there is an edge in the reduced graph from component $C$ to component $C'$ exactly if in the original graph there is an edge $(u, v)$ where $u \in C$ and $v \in C'$.

**Step 3.** $A_{\text{Inf-DFA}}$ checks whether any path from the start vertex to a final vertex includes a marked vertex (and thus can contain a cycle drawn from the corresponding strong component).

**Step 3.1.** By means of any graph traversal procedure (e.g. DFS, BFS), $A_{\text{Inf-DFA}}$ determines which marked vertices are reachable from the start vertex. It doubly marks these vertices.

**Step 3.2.** By means of a second traversal, $A_{\text{Inf-DFA}}$ determines whether any final vertices can be reached from the doubly marked vertices. If so, there is a path with a cycle from the

---

\(^1\)Recall that a strong component is a maximal set $C$ of vertices, such that for every pair of vertices $u, v \in C$, there are directed paths from $u$ to $v$ and from $v$ to $u$. 
start vertex to a final vertex in \( M \)'s graph, and then \( A_{\text{Inf-DFA}} \) outputs “Recognize”; otherwise \( A_{\text{Inf-DFA}} \) outputs “Reject.”

**Example 4.3.13.** All-DFA = \{\langle M \rangle | L(M) = \Sigma^* \text{ where } \Sigma \text{ is } M \text{'s input alphabet} \}.

\[ L(M) = \Sigma^* \text{ exactly if } L(M) = \phi. \] So to test if \( L(M) = \Sigma^* \), the decision algorithm \( A_{\text{All-DFA}} \) simply constructs the encoding \( \langle M \rangle \) and then uses the program \( P_{\text{Empty-DFA}} \) to test if \( L(M) = \phi \), where \( M \) is the DFA recognizing \( L \), and then outputs the same answer, namely:

\[ A_{\text{All-DFA}}(\langle M \rangle) = \begin{cases} \text{“Recognize”} & \text{if } P_{\text{Empty-DFA}}(\langle M \rangle) = \text{“Recognize”} \\ \text{“Reject”} & \text{if } P_{\text{Empty-DFA}}(\langle M \rangle) = \text{“Reject”} \end{cases} \]

### 4.4 Decidability of Context Free Language Properties

**Example 4.4.1.** Rec-CFG = \{\langle G, w \rangle | G \text{ is a CFG which can generate } w \}.

**Claim 4.4.2.** Rec-CFG is decidable.

*Proof.* The following algorithm \( A_{\text{Rec-CFG}} \) decides Rec-CFG.

**Step 1.** \( A_{\text{Rec-CFG}} \) converts \( G \) to a CNF grammar \( \tilde{G} \), with start symbol \( S \).

**Step 2.** If \( w = \lambda \), \( A_{\text{Rec-CFG}} \) checks if \( S \to \lambda \) is a rule of \( \tilde{G} \) and if so outputs “Recognize” and otherwise outputs “Reject.”

**Step 3.** If \( w \neq \lambda \), the derivation of \( w \) in \( \tilde{G} \), if there is one, would take \( 2|w| - 1 \) steps. \( A_{\text{Rec-CFG}} \) simply generates, one by one, all possible derivations in \( \tilde{G} \) of length \( 2|w| - 1 \). If any of them yield \( w \) then it outputs “Recognize” and otherwise it outputs “Reject.” (This is not intended to be an efficient algorithm.)

**Example 4.4.3.** Empty-CFG = \{\langle G \rangle | G \text{ is a CFG and } L(G) = \phi \}.

**Claim 4.4.4.** Empty-CFG is decidable.

*Proof.* Note that \( L(G) \neq \phi \) if and only if \( G \)'s start variable can generate a string in \( T^* \), where \( T \) is \( G \)'s terminal alphabet. We simply determine this property for each variable \( A \) in \( G \); can \( A \) generate a string in \( T^* \)? This can be done by means of the following algorithm \( A_{\text{Empty-CFG}} \), which marks each such variable.

**Step 1.** \( A_{\text{Empty-CFG}} \) converts the grammar to CNF form (this just simplifies the rest of the description).

**Step 2.** \( A_{\text{Empty-CFG}} \) marks each variable \( A \) for which there is a rule \( A \to a \) or \( A \to \lambda \) (the latter could apply only to \( S \), the start variable).

**Step 3.** Iteratively, \( A_{\text{Empty-CFG}} \) marks each variable \( A \) such that there is a rule \( A \to BC \) and \( B \) and \( C \) are already marked. (We leave an efficient implementation to the reader). \( A_{\text{Empty-CFG}} \) stops when no more variables can be marked.

**Step 4.** \( A_{\text{Empty-CFG}} \) outputs “Reject” if \( S \) is marked and “Recognize” otherwise.
Next, we describe a more efficient algorithm $A_{\text{Eff-Rec-CFG}}$, for determining if a CNF grammar $G$ can generate a string $w$. It runs in time $O(mn^3)$, where $m$ is the number of rules in $G$ and $n = |w|$.

First, we introduce a little notation. Let $w = w_1w_2\cdots w_n$, where each $w_i \in T$, the terminal alphabet, for $1 \leq i \leq n$. $w_i^l$ denotes the length $l$ substring of $w$ beginning at $w_i$: $w_i^l = w_i w_{i+1} \cdots w_{i+l-1}$.

$A_{\text{Eff-Rec-CFG}}$ uses dynamic programming. Specifically, in turn, for $l = 1, 2, \cdots, n$, it determines, for each variable $A$, whether $A$ can generate $w_i^l$, for each possible value of $i$, i.e. for $1 \leq i \leq n - l + 1$. This information suffices, for $G$ can generate $w$ exactly if $S \Rightarrow^* w_1^n$, when $S$ is $G$'s start variable.

For $l = 1$, the test amounts to asking whether $A \to w_i$ is a rule.

For $l > 1$, the test amounts to the following question:

Is there a rule $A \to BC$, and a length $k$, with $1 \leq k < l$, such that $B$ generates the length $k$ substring of $w$ beginning at $w_i$ and such that $C$ generates the remainder of $w_i^l$ (i.e. $B \Rightarrow^* w_i^k$ and $C \Rightarrow^* w_i^{l-k}$). Note that the results of the tests involving $B$ and $C$ have have already been computed, so for a single rule and a single value of $k$, this test runs in $O(1)$ time.

Summing the running times over all possible values of $i, k, l$, and all $m$ rules yields the overall running time of $O(mn^3)$.

This shows that:

**Lemma 4.4.5.** The decision procedure for language Rec-CFG runs in time $O(mn^3)$ on input $(G, w)$, where $n = |w|$ and $m$ is the number of rules in $G$.

**Example 4.4.6.** Inf-CFG = \{ $G \mid G$ is a CNF grammar and $L(G)$ is infinite \}.

**Claim 4.4.7.** Inf-CFG is decidable.

**Proof.** Note that $L(G)$ is infinite exactly if there is a path in a derivation tree with a repeated variable. The following algorithm, $A_{\text{inf-CFG}}$ identifies the variables that can be repeated in this way; $L(G)$ is infinite exactly if there is at least one such variable. $A_{\text{inf-CFG}}$ proceeds in several steps.

**Step 1.** This step identifies useful variables, variables that can generate non-empty strings of terminals.

This can be done using a marking procedure, First, $A_{\text{inf-CFG}}$ marks the variables $A$ for which there is a rule of the form $A \to a$. Then, iteratively, for each rule $A \to BC$, if both $B$ and $C$ are marked, it also marks $A$, continuing until no additional variables can be marked. The marked variables are exactly the useful variables.

If the start variable $S$ is not useful, the algorithms stops, answering “Reject”. Otherwise, it continues with Step 2.

**Step 2.** Let $U$ be the set of $G$’s useful variables. $A_{\text{inf-CFG}}$ now identifies the reachable useful variables, i.e. those useful variables for which there is a derivation $S \Rightarrow^* \sigma A\tau$, where
\( \sigma, \tau \in U^* \). This is done via the following marking process.

**Step 2.0.** \( A_{\text{Inf-CFG}} \) removes all rules that contain one or more useless (non-useful) variables.

**Step 2.1.** \( A_{\text{Inf-CFG}} \) marks \( S \).

**Step 2.2.** For each unprocessed marked variable \( A \), \( A_{\text{Inf-CFG}} \) marks all variables on the RHS of a rule with \( A \) on the LHS.

When this process terminates, the marked variables are exactly the reachable useful variables.

**Step 3.** Finally, \( A_{\text{Inf-CFG}} \) identifies the repeating, reachable useful variables, namely the variables that can repeat on a derivation tree path.

To do this, \( A_{\text{Inf-CFG}} \) uses a procedure analogous to the one used in Step 2: For each reachable useful variable \( A \), \( A_{\text{Inf-CFG}} \) determines the variables reachable from \( A \); if this collection includes \( A \), then \( A \) is repeating. (Note that Step 2 found the useful variables reachable from \( S \).)

\( A_{\text{Inf-CFG}} \) answers “Recognize” exactly if some variable is repeating.

### 4.5 Undecidability

The *Barber of Seville* is a classic puzzle. The barber of Seville is said to shave all men who do not shave themselves. So who shaves the barber of Seville? To make this into a puzzle the words have to be treated unduly logically. In particular, one has to interpret it to mean that anyone shaved by the barber of Seville does not shave himself. Then if the barber of Seville shaves himself it is because he does not shave himself and in turn this is because he does shave himself.

One way out of this conundrum occurs if the Barber of Seville is a woman. But our purpose here is to look at how to set up this type of conundrum, or contradiction.

![Figure 4.1: Who Shaves the Barber of Seville?](image)

Let us form a table, on one side listing people in their role as people who are shaved, on the other as potential shavers (or barbers). For simplicity, we just name the people 1, 2, \( \cdots \). So the entries in row \( i \) show who is shaved by person \( i \), with entry \((i, j)\) being \( Y \) (“yes”) if person \( i \) shaves person \( j \) and \( N \) (“no”) otherwise. Let row \( b \) be the row for the barber of
Seville. Then, for every \( j \), entries \((b,j)\) and \((j,j)\) are opposite (one Y and one N). This leads to a contradiction for entry \((b,b)\) cannot be opposite to itself. See Figure 4.1.

Now we are ready to show that the halting problem is undecidable by means of a similar argument. We define the halting problem function \( H \) as follows.

\[
H(P, w) = \begin{cases} 
  \text{"Recognize"} & \text{if program } P \text{ halts on input } w \\
  \text{"Reject"} & \text{if program } P \text{ does not halt on input } w
\end{cases}
\]

By halt, we mean that a program completes its computation and stops.

Recall that the program \( P \) and its input are encoded as binary strings. Let \( i \) denote the \( i \)th such string in lexicographic order, namely the order \( 0, 1, 00, 01, 10, 11, 000, \ldots \). Let \( P_i \) denote the \( i \)th string when it is representing a program, and \( w_i \) denote it when it is representing an input. When considering the pair \((P_i, w_j)\), if \( P_i \) is not an encoding of a legal program, or if \( w_j \) is not an encoding of a legal input to \( P_i \), then the output of program \( P_i \) on input \( w_j \) is deemed to be “Halting”.

The output of program \( H \) is illustrated in figure 4.2. By analogy with the Barber of Seville, our aim is to create a new program which has output the opposite of the entries on the diagonal of the output table for \( H \). Program \( P_d \) in Figure 4.2, if it exists, provides this opposite.

More precisely, we define \( D \) as follows.

\[
D(P_i) = \begin{cases} 
  \text{output } \text{"Done"} & \text{if } P_i \text{ runs forever on input } w_i \\
  \text{loop forever} & \text{if } P_i \text{ on input } w_i \text{ eventually halts}
\end{cases}
\]

If \( H \) is computable then clearly there is a program \( P_d \) to compute \( D \). But then, as with the Barber of Seville, \( P_d(w_d) \) is not well defined, for:

\[
P_d(w_d) = \begin{cases} 
  \text{output } \text{"Done"} & \text{if } P_d \text{ runs forever on input } w_d \\
  \text{loop forever} & \text{if } P_d \text{ on input } w_d \text{ halts}
\end{cases}
\]

But this is a contradiction. We have shown:

**Lemma 4.5.1.** There is no program to compute the function \( D \).

We have also shown:

**Theorem 4.5.2.** There is no (2-input) program that computes \( H \), the halting function.
The technique we have just used is called diagonalization. It was first developed to show that the real numbers are not countable. We will show this result next.

Definition 4.5.3. A set \( A \) is listable or countable if it is finite or if there is a function \( f \) such that \( A = \{ f(1), f(2), \cdots \} \).

In other words, \( A \) can be listed by the function \( f \) (\( f(1) = a_1 \) is the first item in \( A \), \( f(2) = a_2 \) is the second item, and so on).

If this listing can be carried out by a program, \( A \) is said to be effectively or recursively enumerable. As we will see later, the halting set \( H = \{ \langle P, w \rangle \mid P \text{ halts on input } w \} \) is recursively enumerable.

![Figure 4.3: \( d \) is not in the listing of reals.](image)

Lemma 4.5.4. The real numbers are not countable.

Proof. We will show that the real numbers in the range \([0, 1]\) are not countable. This suffices to show the result, for given a listing of all the real numbers, one could go through the list, forming a new list of the reals in the range \([0, 1]\).

For a contradiction, suppose that there were a listing of the real numbers in the range \([0, 1]\), \( r_1, r_2, \cdots \), say. Imagine that each real number is provided as an infinite decimal: \( r_i = 0.r_{i1}r_{i2} \cdots \), when each \( r_{ij} \) is a digit (\( r_{ij} \in \{0, \cdots, 9\} \)). Note that \( 1 = 0.999 \cdots \). This could mean that each real \( r \) is presented using a program \( P \), that on input \( i \) generates the \( i \)th digit of \( r \).

We create a new decimal, \( d = d_1d_2\cdots \), such that \( d \in [0, 1] \) yet \( d \neq r_i \) for all \( i \); so the listing of reals in the range \([0, 1]\) would not include \( d \); but it must do so as it is a listing of all the reals in this range. This is a contradiction, which shows that the reals are not countable.

It remains to define \( d \), which we do as follows:

\[
d_i = r_{ii} + 2 \mod 10 \quad \text{for all } i.
\]

Suppose that \( d = r_j \) for some \( j \). As \( d_j \neq r_{jj} \), the only way \( d \) and \( r_j \) could be equal is if one of them had the form \( 0.sx99 \cdots \) and the other had the form \( 0.s(x + 1)00 \cdots \), where \( x \) is a
single digit and $s$ is a string of digits. But as the shift on the $j$th digit is by 2 this is not possible.

Again, this is a construction by diagonalization, with the fact that $0.99 \cdots = 1.00 \cdots$ creating a small complication. The construction of $d$ is illustrated in Figure 4.3.

By contrast the rationals are countable. By a rational we mean a ratio $a/b$ where $a$ and $b$ are positive integers, but not necessarily coprime. So for the purposes of listing the rationals we will view $a/b$ and $2a/2b$ as distinct rationals (it is a simple exercise to modify the listing function to eliminate such duplicates).

\begin{figure}[h]
\centering
\begin{tikzpicture}
  \draw[->] (0,0) -- (0,3) node[above] {\ldots a};
  \draw[->] (0,0) -- (3,0) node[right] {\ldots b};
  \foreach \i in {1,2,3,4} \draw (\i,0) node {$\ldots$};
  \foreach \i in {1,2,3} \draw (0,\i) node {$\ldots$};
  \draw (1,1) -- (2,2) -- (3,3) -- (4,4);
\end{tikzpicture}
\caption{Listing the Rationals.}
\end{figure}

**Lemma 4.5.5.** The rationals are countable.

*Proof.* The rationals can be displayed in a 2-D table, as shown in Figure 4.4. The rows and columns are indexed by the positive integers, and entry $(a, b)$ represents rational $a/b$. So the task reduces to listing the table entries, which is done by going through the forward diagonals, one by one, in the order of increasing $a + b$. That is, first the entry with $a + b$ value 2 is listed, namely the entry (1,1); then the entries with $a + b$ value 3 are listed, namely the entries (2,1), (1,2); next the entries with $a + b$ value 4 are listed, namely the entries (3,1), (2,2), (1,3); and so forth. Within a diagonal, the entries are listed in the order given by increasing the second coordinate.

Clearly every rational is listed eventually. ($(a, b)$ will be the $(a + b - 1)(a + b - 2)/2 + b$th item listed, in fact.) Also every rational is listed exactly once. Thus the rationals are countable.

The same idea can be used to list the items in a $d$-dimensional table where each coordinate is indexed by the positive integers. Let the coordinate names be $x_1$, $x_2$, \ldots, $x_d$, respectively. Then, in turn, the entries with $x_1 + x_2 + \cdots + x_d = k$, for $k = d, d + 1, d + 2, \cdots$ are listed. For a given value of $k$, in turn, the entries with $x_d = 1, 2, \cdots, k - d + 1$ are listed recursively.

So for $d = 3$ the listing begins (1,1,1), (2,1,1), (1,2,1), (1,1,2), (3,1,1), (2,2,1), (1,3,1), (2,1,2), etc. We call this the diagonal listing, and we use it to show that $H$ is recursively enumerable.
Lemma 4.5.6. \( H \) is recursively enumerable.

Proof. The listing program explores a 3-dimensional table in the diagonal listing order. At the \((i, j, k)\)th table entry, it simulates program \( P_i \) on input \( w_j \) for \( k \) steps. If \( P_i(w_j) \) runs to completion in exactly \( k \) steps then the listing program outputs the encoding \( \langle i, j \rangle \) of the pair \( (i, j) \).

Clearly, if \( P_i \) halts on input \( w_j \), then \( \langle i, j \rangle \) occurs in this listing, and further it occurs exactly once. As this listing is produced by a program it follows that \( H \), the set listed, is recursively enumerable.

Next, we relate recursive enumerability and decidability.

Lemma 4.5.7. If \( L \) and \( \overline{L} \) are both recursively enumerable then \( L \) is decidable.

Proof. It suffices to give an algorithm \( A_L \) to decide \( L \). \( A_L \) will use the listing procedures for \( L \) and \( \overline{L} \). Let \( \text{List}_L(x) \) be the program that on input \( x \) returns the \( x \)th item in a listing of \( L \), and let \( \text{List}_{\overline{L}}(x) \) be the analogous program with respect to \( \overline{L} \). Then, on input \( w \), \( A_L \) simply runs \( \text{List}_L(x) \) and \( \text{List}_{\overline{L}}(x) \) for increasing values of \( x \) (\( x = 1, 2, \cdots \) in turn) until one of them returns the value \( w \), thereby showing in which set \( w \) occurs. At this point, \( A_L \) outputs “Recognize” or “Reject”, as appropriate. In more detail, \( A_L \) is the following program.

\[
A_L(w):
\]

\[
\begin{align*}
\text{found } & \leftarrow \text{False}; \\
\text{x } & \leftarrow 1 \\
\text{while } \not\text{(found)} \text{ do} \\
\quad w_L & \leftarrow \text{List}_L(x) \\
\quad w_{\overline{L}} & \leftarrow \text{List}_{\overline{L}}(x) \\
\quad \text{if } w = w_L \text{ or } w = w_{\overline{L}} \\
\quad \quad & \text{then found } \leftarrow \text{True} \\
\quad \quad \text{else } x \leftarrow x + 1 \\
\text{end while} \\
\quad \text{if } w = w_L \text{ then return(“Recognize”) } \\
\quad \text{else return(“Reject”) }
\end{align*}
\]

Note that \( w \) occurs in one of the lists \( L \) or \( \overline{L} \). Consequently, \( w \) must be listed eventually, and therefore the loop in the above program also terminates eventually (when it reaches the item \( w \) in the list containing \( w \)).

We can now show that \( \overline{H} \) is not recursively enumerable.

Lemma 4.5.8. \( \overline{H} \) is not recursively enumerable.

Proof. Recall that \( H \) is recursively enumerable (by Lemma 4.5.6). Were \( \overline{H} \) also recursively enumerable, then, by Lemma 4.5.7, \( H \) would be decidable, which is not the case (by Theorem 4.5.2). This shows that \( \overline{H} \) cannot be recursively enumerable. (Strictly, this was a proof by contradiction.)

Notice that membership and non-membership are not symmetric in a recursively enumerable but non-decidable set such as \( H \). While membership can be demonstrated simply by listing the set and encountering the item, there is no test for non-membership.
4.6 Undecidability via Reductions

We turn to showing undecidability via reductions. The form of the argument will be as follows.

Suppose that we are given an algorithm (or program) $A_L$ which is claimed to decide set $L$. Suppose that using $A_L$ as a subroutine, we create another algorithm $A_J$ to decide set $J$. But suppose that we already know set $J$ to be undecidable, for example if $J = H$, the halting set. We can then conclude that $A_L$ does not decide set $L$, and in fact that there is no algorithm to decide set $L$. The latter claim follows by a proof by contradiction: assume that there is such an algorithm, and call it $A_L$; then the previous argument shows that $A_L$ does not decide $L$, a contradiction.

**Example 4.6.1.** $W = \{\langle Q \rangle | Q \text{ on input string 0 eventually halts}\}$.

**Lemma 4.6.2.** If there is an algorithm $A_W$ to decide $W$, then there is also an algorithm to decide $H$, the halting set.

**Proof.** Here is the algorithm $A_H$ deciding $H$. It will use $A_W$ as a subroutine.

On input $\langle P, w \rangle$:

$A_H$ will build (compute the encoding of) a one-input program $R_{P,w}$ which it then inputs to the algorithm $A_W$. $A_H$ concludes by reporting the result of running $A_W$ on input $\langle R_{P,w} \rangle$.

This means that we want $A_W$ to behave as follows:

$$A_H(\langle P, w \rangle) = A_W(\langle R_{P,w} \rangle) = \begin{cases} 
\text{"Recognize"} & \text{if } P \text{ eventually halts on input } w \\
\text{"Reject"} & \text{if } P \text{ does not halt on input } w 
\end{cases}$$

Now by the definition of $W$:

$$A_W(\langle R_{P,w} \rangle) = \begin{cases} 
\text{"Recognize"} & \text{if } R_{P,w}(0) \text{ halts} \\
\text{"Reject"} & \text{if } R_{P,w}(0) \text{ does not halt}
\end{cases}$$

This means that we want:

$$R_{P,w}(0) \text{ halts if } P \text{ eventually halts on input } w$$

$$R_{P,w}(0) \text{ does not halt if } P \text{ does not halt on input } w.$$  

Here is a program $R_{P,w}$ that meets the above requirement:

$$R_{P,w}(x)$$

Run $P$ on input $w$ (note that $R_{P,w}$ ignores its input).

---

2The notation $R_{P,w}$ is meant to indicate that program $R$ is determined in part by $P$ and $w$. (Thus $P$ might be a subroutine in $R$, and $w$ might be the initial value of one of $R$'s variables. $w$ is not an input to $R$. $x$ denotes $R$'s input.)
Clearly, for any $x$, $R_{P,w}(x)$ halts if $P$ eventually halts on input $w$, and $R_{P,w}(x)$ does not halt if $P$ does not halt on input $w$, and hence it does so for $x = 0$ in particular.

Thus our algorithm for deciding $H$ proceeds as follows:

**Step 1.** $A_H$ computes $\langle R_{P,w} \rangle$, the encoding of program $R_{P,w}$.

**Step 2.** $A_H$ simulates algorithm $A_W$ on input $\langle R_{P,w} \rangle$ and outputs the result of $A_W(\langle R_{P,w} \rangle)$.

**Corollary 4.6.3.** $W$ is undecidable, that is there is no algorithm to decide $W$.

**Example 4.6.4.** Let $X = \{ \langle Q \rangle \mid \text{either $Q$ on input 0 eventually halts, or $Q$ on input 1 eventually halts, or both} \}$.

**Lemma 4.6.5.** If there is an algorithm $A_X$ to decide $X$, then there is also an algorithm $A_H$ to decide $H$.

**Proof.** This proof is very similar to the previous one. Again, $A_H$ will build a program $R'_{P,w}$ with the following characteristics:

$$A_X(\langle R'_{P,w} \rangle) = \begin{cases} \text{"Recognize"} & \text{if $P$ eventually halts on input $w$} \\
\text{"Reject"} & \text{if $P$ does not halt on input $w$} \end{cases}$$

$A_H$ then simulates $A_X(\langle R'_{P,w} \rangle)$, reporting the result of this simulation as its output.

This means that we want:

- at least one of $R'_{P,w}(\langle 0 \rangle)$ and $R'_{P,w}(\langle 1 \rangle)$ halts if $P$ eventually halts on input $w$
- both $R'_{P,w}(\langle 0 \rangle)$ and $R'_{P,w}(\langle 1 \rangle)$ do not halt if $P$ does not halt on input $w$

It is easy to check that $R'_{P,w} = R_{P,w}$ meets this requirement. Thus we can use the following algorithm $A_H$ to decide $H$.

On input $\langle P, w \rangle$:

**Step 1.** $A_H$ constructs the encoding $\langle R_{P,w} \rangle$.

**Step 2.** $A_H$ simulates $A_X(\langle R_{P,w} \rangle)$ and give its result as the output of $A_H$. $\square$

**Corollary 4.6.6.** $X$ is undecidable.

**Example 4.6.7.** $Y = \{ \langle Q \rangle \mid \text{there is a variable $v$ in $Q$ that is never assigned a value when $Q$ is run on input 1} \}$.

**Lemma 4.6.8.** If there is an algorithm $A_Y$ to decide $Y$, then there is also an algorithm $A_H$ to decide $H$.

**Proof.** Here is the algorithm $A_H$ to decide $H$.

On input $\langle P, w \rangle$:

$A_H$ will compute the encoding of a program $R''_{P,w}$ which it then inputs to $A_Y$. What we want is for:

$$A_H(\langle P, w \rangle) = A_Y(\langle R''_{P,w} \rangle) = \begin{cases} \text{"Recognize"} & \text{if $P$ eventually halts on input $w$} \\
\text{"Reject"} & \text{if $P$ does not halt on input $w$} \end{cases}$$

This means that we want:
4.6. UNDECIDABILITY VIA REDUCTIONS

- If \( P \) eventually halts on input \( w \) then, when \( R''_{P,w} \) is run on input 1, \( R''_{P,w} \) has a variable that is never assigned a value, and

- If \( P \) does not halt on input \( w \) then, when \( R''_{P,w} \) is run on input 1, every variable of \( R''_{P,w} \) is assigned a value.

Let’s try the following program for \( R''_{P,w} \):

**On input** \( x \):

**Step 1.** \( R''_{P,w} \) simulates \( P(w) \).

**Step 2.** For every variable \( z \), that appears in \( P \) or the simulation environment, do: \( z \leftarrow 0 \).

**Step 3.** \( v \leftarrow 1 \), where \( v \) is a variable that does not appear in \( P \) or the simulation environment.

When \( R''_{P,w} \) is run on input \( x \) and \( x = 1 \) in particular, if \( P \) halts on input \( w \), then every variable appearing in \( R''_{P,w} \) is assigned a value, while if \( P \) does not halt on input \( w \), at the very least, variable \( v \) in \( R''_{P,w} \) is not assigned a value.

Oops, this is back to front. Unfortunately, this is unavoidable. So let’s change our goal for \( A_Y \). Let’s require:

\[
A_Y(\langle R''_{P,w} \rangle) = \begin{cases} 
"Reject" & \text{if } P \text{ eventually halts on input } w \\
"Recognize" & \text{if } P \text{ does not halt on input } w 
\end{cases}
\]

so \( A_H(\langle P, w \rangle) = \text{Opposite}(A_Y(\langle R''_{P,w} \rangle)) \).

This means that we want:

- If \( P \) eventually halts on input \( w \) then, when \( R''_{P,w} \) is run on input 1, every variable of \( R''_{P,w} \) is assigned a value.

- If \( P \) does not halt on input \( w \) then, when \( R''_{P,w} \) is run on input 1, \( R''_{P,w} \) has a variable that is never assigned a value.

But this is achieved by the above program \( R''_{P,w} \).

Now, our algorithm \( A_H \) for deciding \( H \) simply reports the opposite answer to \( A_Y(\langle R''_{P,w} \rangle) \). So the algorithm is the following:

**Step 1.** \( A_H \) constructs the encoding \( \langle R''_{P,w} \rangle \).

**Step 2.** \( A_H \) simulates \( A_Y(\langle R''_{P,w} \rangle) \).

**Step 3.** \( A_H \) reports the opposite answer to that given by \( A_Y \) in Step 2.

\[ \square \]

**Corollary 4.6.9.** \( Y \) is undecidable.

**Example 4.6.10.** Never-Halt = \{ \langle Q \rangle \mid Q \text{ does not halt on any input} \}.

**Lemma 4.6.11.** If there is an algorithm \( A_{NH} \) to decide Never-Halt, then there is also an algorithm \( A_H \) to decide \( H \).
Proof. The algorithm $A_H$ deciding $H$ will use $A_{NH}$ as a subroutine. On input $\langle P, w \rangle$, $A_H$ will compute the encoding of a one-input program $R^3_{P,w}$ which it then inputs to the algorithm $A_{NH}$. What we want is that:

$$A_H(\langle P, w \rangle) = A_{NH}(\langle R^3_{P,w} \rangle) = \begin{cases} "Recognize" & \text{if } P \text{ eventually halts on input } w \\ "Reject" & \text{if } P \text{ does not halt on input } w \end{cases}$$

Now by the definition of Never-Halt:

$$A_{NH}(\langle R^3_{P,w} \rangle) = \begin{cases} "Recognize" & \text{if } R^3_{P,w} \text{ never halts} \\ "Reject" & \text{if } R^3_{P,w} \text{ halts on some input} \end{cases}$$

This means that we want:

$$R^3_{P,w} \text{ never halts if } P \text{ eventually halts on input } w$$
$$R^3_{P,w} \text{ halts on some input if } P \text{ does not halt on input } w.$$

It does not seem possible to build such an $R$. Let’s try switching the outputs given by $A_{NH}$ to be:

$$A_{NH}(\langle R^3_{P,w} \rangle) = \begin{cases} "Reject" & \text{if } P \text{ eventually halts on input } w \\ "Recognize" & \text{if } P \text{ does not halt on input } w \end{cases}$$

so $A_H(\langle P, w \rangle) = \text{Opposite}(A_{NH}(\langle R^3_{P,w} \rangle))$.

And then we want:

$$R^3_{P,w} \text{ never halts if } P \text{ does not halt on input } w$$
$$R^3_{P,w} \text{ halts on some input if } P \text{ eventually halts on input } w.$$

The program $R^3_{P,w} = R_{P,w}$ from Example 4.6.2 meets the above requirement.

Thus our algorithm for deciding $H$ proceeds as follows:

**Step 1.** $A_H$ computes $\langle R_{P,w} \rangle$, the encoding of program $R_{P,w}$.

**Step 2.** $A_H$ simulates algorithm $A_{NH}(\langle R_{P,w} \rangle)$ and outputs the opposite of its result. \( \square \)

**Corollary 4.6.12.** Never-Halt is undecidable.

**Example 4.6.13.** Equal-Prog = $\{\langle Q_1, Q_2 \rangle \mid Q_1 \text{ and } Q_2 \text{ halt on exactly the same inputs} \}$. We write $Q_1 = Q_2$ if $\langle Q_1, Q_2 \rangle \in \text{Equal-Prog}$ and $Q_1 \neq Q_2$ if $\langle Q_1, Q_2 \rangle \notin \text{Equal-Prog}$, for short.

**Lemma 4.6.14.** If there is an algorithm $A_{EP}$ to decide Equal-Prog, then there is also an algorithm $A_{NH}$ to decide Never-Halt.

**Proof.** The algorithm $A_{NH}$ deciding Never-Halt will use $A_{EP}$ as a subroutine. On input $\langle P \rangle$, $A_{NH}$ will compute the encoding of programs $R^1_P$ and $R^5_P$ which it then inputs to the algorithm $A_{EP}$. What we want is that:

$$A_{NH}(\langle P \rangle) = A_{EP}(\langle R^1_P, R^5_P \rangle) = \begin{cases} "Recognize" & \text{if } P \text{ does not halt on any input} \\ "Reject" & \text{if } P \text{ halts on some input} \end{cases}$$
Now by the definition of Equal-Prog:

\[ A_{EP}(\langle R_4^P, R_5^P \rangle) = \begin{cases} 
"Recognize" & \text{if } R_4^P = R_5^P \\
"Reject" & \text{if } R_4^P \neq R_5^P 
\end{cases} \]

This means that we want:

\[ R_4^P = R_5^P \text{ if } P \text{ does not halt on any input} \]
\[ R_4^P \neq R_5^P \text{ if } P \text{ halts on some input.} \]

This suggests the following choice for \( R_4^P \) and \( R_5^P \). Set \( R_4^P = P \) and \( R_5^P = N \), a program that never halts. Here is \( N(x) \): loop forever. This pair does meet the above criteria. This yields the following algorithm \( A_{Nh} \) for deciding \textit{Never-Halt}.

\[ A_{Nh} \text{ simulates algorithm } A_{EP}(\langle P, N \rangle) \text{ and outputs its result. (Note that } \langle P \rangle \text{ is the input to } A_{Nh}, \text{ and } \langle N \rangle \text{ can be stored as the initial value of one of the variables of } A_{Nh}. \) \]

\[ \square \]

**Corollary 4.6.15.** Equal-Prog is undecidable.

**Exercises**

1. Let \( \text{Rec-GNFA} = \{ \langle M, w \rangle \mid M \text{ is a GNFA and } w \in L(M) \} \). Show that \( \text{Rec-GNFA} \) is decidable.

2. i. Show that \( R \subseteq S \) if and only if \( R \cap \overline{S} = \emptyset \).

   ii. Let \( \text{Reg-Contain} = \{ \langle M_R, M_S \rangle \mid M_R \text{ and } M_S \text{ are DFAs recognizing regular languages } R \text{ and } S \text{ respectively, and } R \subseteq S \} \).

   Show that \( \text{Reg-Contain} \) is decidable.

   Hint: Use a reduction to \( \text{Empty-DFA} \).

3. Let \( \text{Eq-DFA-NFA} = \{ \langle M, N \rangle \mid M \text{ is a DFA and } N \text{ is an NFA with } L(M) = L(N) \} \). Show that \( \text{Eq-DFA-NFA} \) is decidable.

4. Let \( \text{Eq-Rev} = \{ \langle M \rangle \mid M \text{ is a DFA, } L = L(M), M^R \text{ is an NFA recognizing } L^R, \text{ and } L(M) = L(M^R) \} \).

   Show that \( \text{Eq-Rev} \) is decidable. You may assume the result of Chapter 2, No. 10.

5. Let \( \text{Pumpable} = \{ \langle M \rangle \mid M \text{ is a DFA, and for every } w \in L(M), w \text{ is pumpable; i.e. } w \text{ can be written as } w = xz, \text{ and there is a non-empty string } y \text{ such that } xy^iz \in L(M) \text{ for every integer } i \geq 0 \} \).

   Show that \( \text{Pumpable} \) is decidable.

   Hint. What can you say about \( M \)'s graph if every string it recognizes is pumpable? Your algorithm needs to test this property.
6. Let \(\text{CFL-Int-}a^* = \{\langle G \rangle \mid G \text{ is a context free grammar, and } L(G), \text{ the language it generates, satisfies } L(G) \cap a^* \neq \emptyset}\}.

Show that CFL-Int-\(a^*\) is decidable.
Hint. Use a reduction to Empty-CFL.

7. \(A = \{\langle G \rangle \mid G \text{ is a CNF grammar with start variable } S, \text{ terminal alphabet } \{a,b\}, \text{ and there is a string } x \in a^* \text{ such that } S \Rightarrow^* x\}\).
That is, there is a string containing only \(a\)'s in \(L(G)\).
Show that \(A\) is decidable.

8. Let \(\text{Inf-PDA} = \{\langle M_L \rangle \mid M_L \text{ is a PDA recognizing language } L, \text{ and } L \text{ is infinite}\}\).
Show that \(\text{Inf-PDA}\) is decidable.

9. Let \(\text{CFL-Int-Reg} = \text{CFL-IR} = \{\langle G, M \rangle \mid G \text{ is a context free grammar, } M \text{ is a DFA and } L(G) \cap L(M) \neq \emptyset\}\}.
Show that CFL-IR is decidable.
Hint. Use a reduction to Empty-CFL.

10. Let \(L(x)\) be a computable function that lists *always halting integer output programs*: a program is an always halting integer output program if for every possible input it eventually outputs an integer.

So \(L\) is computed by a program, \(P_L\) say. Specifically, \(L(1), L(2), \ldots = P_{j_1}, P_{j_2}, \ldots\) is a list of always halting integer output programs.

Prove, by diagonalization, that there is an always halting integer output program \(Q\) not on the list generated by \(L\).

11. Let \(A\) and \(B\) be recursively enumerable sets. Suppose that \(\overline{A} \cap \overline{B} = \emptyset\). Show that there is a decidable set \(C\) such that \(\overline{A} \subseteq C\) and \(\overline{B} \subseteq C\).
Hint. Modify the algorithm for deciding membership in set \(L\) if both \(L\) and \(\overline{L}\) are recursively enumerable. (\(A\) and \(B\) will replace \(L\) and \(\overline{L}\); you will need to define \(L\) suitably. This will involve three cases for your membership test: when \(x \in A - B\), when \(x \in B - A\), and when \(x \in A \cap B\). Which case(s) apply when \(x \in \overline{A}\)?)

12. Let \(\text{Halt-Exactly-Once} = \text{HEO} = \{\langle Q \rangle \mid Q \text{ halts on exactly one input}\}\).
Suppose that you are given an algorithm \(A_{\text{HEO}}\) that decides HEO. Using \(A_{\text{HEO}}\) as a subroutine, give an algorithm \(A_H\) to decide \(H\).

13. Let \(\text{Mixed} = \{\langle Q \rangle \mid Q \text{ halts on input 1 and does not halt on input 2}\}\).
Suppose that you are given an algorithm \(A_{\text{Mixed}}\) that decides Mixed. Using \(A_{\text{Mixed}}\) as a subroutine, give an algorithm \(A_H\) to decide \(H\).

14. Let \(\text{Dead-Code} = \text{DC} = \{\langle Q \rangle \mid Q \text{ contains a line of code that is not executed on any input to } Q\}\).
Suppose that you are given an algorithm $A_{DC}$ that decides $DC$. Using $A_{DC}$ as a subroutine, give an algorithm $A_H$ to decide $H$.

Comment: You may assume that the lines of code are numbered consecutively $1, 2, 3, \ldots$, and that there is a command “GoTo line $k$” where $k$ can take on any positive integer value, and which causes the program to continue its execution from line $k$. Further, you may assume that any single line of code consists of a computation which terminates (so, for example, there are no infinite loops in a single line of code).

15. Let $\text{Useless-Var} = UV = \{\langle Q \rangle \mid Q \text{ contains a variable that remains unassigned whatever the input to } Q\}$.

Suppose that you are given an algorithm $A_{UV}$ that decides $UV$. Using $A_{UV}$ as a subroutine, give an algorithm $A_H$ to decide $H$.

16. Let $\text{Even-Length-Halt} = \text{ELH} = \{\langle Q \rangle \mid Q \text{ eventually halts on all input strings of even length}\}$.

Suppose that you are given an algorithm $A_{\text{ELH}}$ that decides $\text{ELH}$. Using $A_{\text{ELH}}$ as a subroutine, give an algorithm $A_{\text{EQ}}$ to decide $\text{EQ}$. Recall that $\text{EQ}$ is the set of all pairs of programs that halt on exactly the same input:

$\text{EQ} = \{\langle Q_1, Q_2 \rangle \mid \text{for all inputs } x, Q_1(x) \text{ eventually halts exactly if } Q_2(x) \text{ eventually halts}\}$.

17. Let $\text{Inf-Mixed} = \text{IM} = \{\langle Q \rangle \mid Q \text{ eventually halts on infinitely many inputs and fails to halt on infinitely many inputs}\}$.

Suppose that you are given an algorithm $A_{\text{IM}}$ that decides $\text{IM}$. Using $A_{\text{IM}}$ as a subroutine, give an algorithm $A_H$ to decide $H$.

18. Let $\text{Sometime-Halt} = \text{SH} = \{\langle Q \rangle \mid Q \text{ halts on at least one input}\}$.

Suppose that you are given an algorithm $A_{\text{SH}}$ that decides $\text{SH}$. Using $A_{\text{SH}}$ as a subroutine, give an algorithm $A_H$ to decide $H$.

19. Let $\text{Mixed} = \{\langle Q \rangle \mid Q \text{ halts on input 0 and does not halt on input 1}\}$.

Suppose that you are given an algorithm $A_{\text{Mixed}}$ that decides $\text{Mixed}$. Using $A_{\text{Mixed}}$ as a subroutine, give an algorithm $A_H$ to decide $H$.

20. Let $\text{Inf-Halt} = \text{IH} = \{\langle Q \rangle \mid Q \text{ halts on infinitely many inputs}\}$.

Suppose that you are given an algorithm $A_{\text{IH}}$ that decides $\text{IH}$. Using $A_{\text{IH}}$ as a subroutine, give an algorithm $A_H$ to decide $H$.

21. Let $\text{Inf-Not-Halt} = \text{INH} = \{\langle Q \rangle \mid Q \text{ fails to halt on infinitely many inputs}\}$.

Suppose that you are given an algorithm $A_{\text{INH}}$ that decides $\text{INH}$. Using $A_{\text{INH}}$ as a subroutine, give an algorithm $A_H$ to decide $H$. 
22. Two programs $P$ and $Q$ are \textit{functionally equivalent} if for every possible input $x$, $P(x) = Q(x)$, where this is taken to mean that if $P(x)$ does not halt then $Q(x)$ does not halt either.

A set $S$ of programs is said to be \textit{functionally defined} if for any pair of functionally equivalent programs $P$ and $Q$, $P \in S$ if and only if $Q \in S$.

e.g. the set $S = \{\langle Q \rangle \mid Q \text{ halts on input 0} \}$ is functionally defined, but the set $T = \{\langle Q \rangle \mid Q \text{ has 10 lines of code} \}$ is not functionally defined.

Rice’s Theorem states that there is no algorithm to decide $S$ if $S$ is a non-trivial functionally defined set (i.e. $S \neq \phi$ and $S \neq \Sigma^*$, where $\Sigma$ is the alphabet used for strings in $S$).

i. Prove that if there is an algorithm $A_S$ to decide some non-trivial functionally defined set $S$ then there is also an algorithm to decide $H$.

Hint. Let $Q_N$ be a program that never halts. Suppose that $Q_N \in S$ (if not, simply switch the roles of $S$ and $\overline{S}$). Let $Q_h$ be a program in $\overline{S}$. Now use $Q_N$ and $Q_h$ to construct a program $R_{P,x}$ such that $R_{P,x} \in S$ exactly if $P(x)$ halts.

ii. By applying Rice’s Theorem, answer Questions 12, 13, 16–21.

iii. Why does Rice’s Theorem not apply to Questions 14 and 15?