5.3: Diagonalization and Undecidable Problems

In this section, we will use a technique called diagonalization to find a natural language that isn't recursively enumerable.

This will lead us to a language that is recursively enumerable but is not recursive.

It will also enable us to prove the undecidability of the halting problem.

Diagonalization

To find a non-r.e. language, we can use diagonalization.

Let Σ be the alphabet used to describe programs: the letters and digits, plus the elements of $\{\langle \mathsf{comma} \rangle, \langle \mathsf{perc} \rangle, \langle \mathsf{tilde} \rangle, \langle \mathsf{openPar} \rangle, \langle \mathsf{closPar} \rangle, \langle \mathsf{less} \rangle, \langle \mathsf{great} \rangle\}$.

As explained in Section 5.1, every element of Σ^* either describes a unique closed program, or describes no closed programs.

Diagonalization

To find a non-r.e. language, we can use diagonalization.

Let Σ be the alphabet used to describe programs: the letters and digits, plus the elements of
$\{\langle\text{comma}\rangle, \langle\text{perc}\rangle, \langle\text{tilde}\rangle, \langle\text{openPar}\rangle, \langle\text{closPar}\rangle, \langle\text{less}\rangle, \langle\text{great}\rangle\}$.

As explained in Section 5.1, every element of Σ^* either describes a unique closed program, or describes no closed programs.

Given $w \in \Sigma^*$, we write $L(w)$ for:

- $\emptyset$, if w doesn't describe a closed program; and
- $L(pr)$, where pr is the unique closed program described by w, if w does describe a closed program.

Diagonalization

To find a non-r.e. language, we can use diagonalization.

Let Σ be the alphabet used to describe programs: the letters and digits, plus the elements of

$\{\langle\mathsf{comma}\rangle, \langle\mathsf{perc}\rangle, \langle\mathsf{tilde}\rangle, \langle\mathsf{openPar}\rangle, \langle\mathsf{closPar}\rangle, \langle\mathsf{less}\rangle, \langle\mathsf{great}\rangle\}$.

As explained in Section 5.1, every element of Σ^* either describes a unique closed program, or describes no closed programs.

Given $w \in \Sigma^*$, we write $L(w)$ for:

- $\emptyset$, if w doesn't describe a closed program; and
- $L(pr)$, where pr is the unique closed program described by w, if w does describe a closed program.

Thus $L(w)$ will always be a set of strings, even though it won't always be a language.

Diagonalization

Consider the infinite table of 0's and 1's in which both the rows and the columns are indexed by the elements of Σ^*, listed in ascending order according to our standard total ordering, and where a cell (w_n, w_m) contains 1 iff $w_n \in L(w_m)$, and contains 0 iff $w_n \notin L(w_m)$.

Each recursively enumerable language is $L(w_m)$ for some (non-unique) m, but not all the $L(w_m)$ are languages.

Diagonalization

Here is how part of this table might look, where w_i, w_j and w_k are sample elements of Σ^*:

	$\cdots$	w_i	$\cdots$	w_j	$\cdots$	w_k	$\cdots$
$\vdots$							
w_i		1		1		0	
$\vdots$							
w_j		0		0		1	
$\vdots$							
w_k		0		1		1	
$\vdots$							

We have that $w_i \in L(w_j)$ and $w_j \notin L(w_i)$.

Diagonalization

To define a non-r.e. Σ-language, we work our way down the diagonal of the table, putting w_n into our language just when cell (w_n, w_n) of the table is 0, i.e., when $w_n \notin L(w_n)$.

With our example table:

- $L(w_i)$ is not our language, since $w_i \in L(w_i)$, but w_i is not in our language;
- $L(w_j)$ is not our language, since $w_j \notin L(w_j)$, but w_j is in our language; and
- $L(w_k)$ is not our language, since $w_k \in L(w_k)$, but w_k is not in our language.

In general, there is no $n \in \mathbb{N}$ such that $L(w_n)$ is our language. Consequently our language is not recursively enumerable.

Diagonalization

We formalize the above ideas as follows. Define languages L_d ("d" for "diagonal") and L_a ("a" for "accepted") by:

$$L_d = \{\, w \in \Sigma^* \mid w \notin L(w) \,\}, \text{ and}$$
$$L_a = \{\, w \in \Sigma^* \mid w \in L(w) \,\}.$$

Thus $L_d = \Sigma^* - L_a$.

We have that, for all $w \in \Sigma^*$, $w \in L_a$ iff $w \in L(pr)$, for some closed program pr (which will be unique) described by w. (When w doesn't describe a closed program, $L(w) = \emptyset$.)

Theorem 5.3.1
L_d *is not recursively enumerable.*

Proof. Suppose, toward a contradiction, that L_d is recursively enumerable. Thus, there is a closed program pr such that $L_d = L(pr)$. Let $w \in \Sigma^*$ be the string describing pr. Thus $L(w) = L(pr) = L_d$.

There are two cases to consider.

□

Theorem 5.3.1

L_d is not recursively enumerable.

Proof. Suppose, toward a contradiction, that L_d is recursively enumerable. Thus, there is a closed program pr such that $L_d = L(pr)$. Let $w \in \Sigma^*$ be the string describing pr. Thus $L(w) = L(pr) = L_d$.

There are two cases to consider.

- Suppose $w \in L_d$. Then $w \notin L(w) = L_d$—contradiction.
- Suppose $w \notin L_d$. Since $w \in \Sigma^*$, we have that $w \in L(w) = L_d$—contradiction.

Since we obtained a contradiction in both cases, we have an overall contradiction. Thus L_d is not recursively enumerable. □

Theorem 5.3.2

L_a *is recursively enumerable.*

Proof. Let acc be the closed program that, when given $\mathbf{str}(w)$, for some $w \in \mathbf{Str}$, acts as follows.

$\square$

Theorem 5.3.2

L_a is recursively enumerable.

Proof. Let acc be the closed program that, when given $\mathbf{str}(w)$, for some $w \in \mathbf{Str}$, acts as follows. First, it attempts to parse $\mathbf{str}(w)$ as a program pr, represented as the value $\overline{pr}$. If this attempt fails, acc returns $\mathbf{const}(\mathrm{false})$.

□

Theorem 5.3.2
L_a is recursively enumerable.

Proof. Let acc be the closed program that, when given $\mathbf{str}(w)$, for some $w \in \mathbf{Str}$, acts as follows. First, it attempts to parse $\mathbf{str}(w)$ as a program pr, represented as the value $\overline{pr}$. If this attempt fails, acc returns $\mathbf{const}(\mathrm{false})$. If pr is not closed, then acc returns $\mathbf{const}(\mathrm{false})$.

□

Theorem 5.3.2

L_a *is recursively enumerable.*

Proof. Let acc be the closed program that, when given $\mathbf{str}(w)$, for some $w \in \mathbf{Str}$, acts as follows. First, it attempts to parse $\mathbf{str}(w)$ as a program pr, represented as the value $\overline{pr}$. If this attempt fails, acc returns $\mathbf{const}(\mathrm{false})$. If pr is not closed, then acc returns $\mathbf{const}(\mathrm{false})$. Otherwise, it uses our interpreter function to evaluate $\mathbf{app}(pr, \mathbf{str}(w))$, using $\overline{\mathbf{app}(pr, \mathbf{str}(w))}$. If this interpretation returns $\overline{\mathbf{const}(\mathrm{true})}$, then acc returns $\mathbf{const}(\mathrm{true})$. If it returns anything other than $\overline{\mathbf{const}(\mathrm{true})}$, then acc returns $\mathbf{const}(\mathrm{false})$. (Thus, if the interpretation never returns, then acc never terminates.)

□

Theorem 5.3.2

L_a is recursively enumerable.

Proof. Let *acc* be the closed program that, when given $\mathbf{str}(w)$, for some $w \in \mathbf{Str}$, acts as follows. First, it attempts to parse $\mathbf{str}(w)$ as a program *pr*, represented as the value $\overline{pr}$. If this attempt fails, *acc* returns $\mathbf{const}(\mathsf{false})$. If *pr* is not closed, then *acc* returns $\mathbf{const}(\mathsf{false})$. Otherwise, it uses our interpreter function to evaluate $\mathbf{app}(pr, \mathbf{str}(w))$, using $\overline{\mathbf{app}(pr, \mathbf{str}(w))}$. If this interpretation returns $\mathbf{const}(\mathsf{true})$, then *acc* returns $\mathbf{const}(\mathsf{true})$. If it returns anything other than $\overline{\mathbf{const}(\mathsf{true})}$, then *acc* returns $\mathbf{const}(\mathsf{false})$. (Thus, if the interpretation never returns, then *acc* never terminates.)

We can check that, for all $w \in \mathbf{Str}$, $w \in L_a$ iff $\mathbf{eval}(\mathbf{app}(acc, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\mathsf{true}))$. Thus L_a is recursively enumerable. □

Corollary 5.3.3

There is an alphabet Σ and a recursively enumerable language $L \subseteq \Sigma^*$ such that $\Sigma^* - L$ is not recursively enumerable.

Proof. $L_a \subseteq \Sigma^*$ is recursively enumerable, but $\Sigma^* - L_a = L_d$ is not recursively enumerable. □

Corollary 5.3.4

There are recursively enumerable languages L_1 and L_2 such that $L_1 - L_2$ is not recursively enumerable.

Proof. Follows from Corollary 5.3.3, since Σ^* is recursively enumerable. □

Corollary 5.3.5

L_a *is not recursive.*

Proof.

Corollary 5.3.5

L_a is not recursive.

Proof. Suppose, toward a contradiction, that L_a is recursive. Since the recursive languages are closed under complementation, and $L_a \subseteq \Sigma^*$, we have that $L_d = \Sigma^* - L_a$ is recursive—contradiction. Thus L_a is not recursive. □

Since $L_a \in$ **RELan** and $L_a \notin$ **RecLan**, we have:

Theorem 5.3.6
The recursive languages are a proper subset of the recursively enumerable languages: **RecLan** $\subsetneq$ **RELan**.

Combining this result with results from Sections 4.8 and 5.1, we have that

RegLan $\subsetneq$ **CFLan** $\subsetneq$ **RecLan** $\subsetneq$ **RELan** $\subsetneq$ **Lan**.

Undecidability of the Halting Problem

We say that a closed program pr *halts* iff $\textbf{eval}\, pr \neq \textbf{nonterm}$.

Theorem 5.3.7

There is no value $halts$ such that, for all closed programs pr,

- *If pr halts, then $\textbf{eval}(\textbf{app}(halts, \overline{pr})) = \textbf{norm}(\textbf{const}(\text{true}))$; and*
- *If pr does not halt, then $\textbf{eval}(\textbf{app}(halts, \overline{pr})) = \textbf{norm}(\textbf{const}(\text{false}))$.*

We say that a closed program pr *halts* iff $\textbf{eval}\, pr \neq \textbf{nonterm}$.

Theorem 5.3.7
There is no value $halts$ such that, for all closed programs pr,

- *If pr halts, then $\textbf{eval}(\textbf{app}(halts, \overline{pr})) = \textbf{norm}(\textbf{const}(\mathrm{true}))$; and*
- *If pr does not halt, then $\textbf{eval}(\textbf{app}(halts, \overline{pr})) = \textbf{norm}(\textbf{const}(\mathrm{false}))$.*

Proof. Suppose, toward a contradiction, that such a $halts$ does exist. We use $halts$ to construct a closed program acc that behaves as follows when run on $\textbf{str}(w)$, for some $w \in \textbf{Str}$. First, it attempts to parse $\textbf{str}(w)$ as a program pr, represented as the value $\overline{pr}$. If this attempt fails, it returns $\textbf{const}(\mathrm{false})$. If pr is not closed, then it returns $\textbf{const}(\mathrm{false})$. Otherwise,

Undecidability of the Halting Problem

We say that a closed program pr *halts* iff $\textbf{eval}\, pr \neq \textbf{nonterm}$.

Theorem 5.3.7

There is no value $halts$ such that, for all closed programs pr,

- *If pr halts, then $\textbf{eval}(\textbf{app}(halts, \overline{pr})) = \textbf{norm}(\textbf{const}(\mathsf{true}))$; and*
- *If pr does not halt, then $\textbf{eval}(\textbf{app}(halts, \overline{pr})) = \textbf{norm}(\textbf{const}(\mathsf{false}))$.*

Proof. Suppose, toward a contradiction, that such a $halts$ does exist. We use $halts$ to construct a closed program acc that behaves as follows when run on $\textbf{str}(w)$, for some $w \in \textbf{Str}$. First, it attempts to parse $\textbf{str}(w)$ as a program pr, represented as the value $\overline{pr}$. If this attempt fails, it returns $\textbf{const}(\mathsf{false})$. If pr is not closed, then it returns $\textbf{const}(\mathsf{false})$. Otherwise, it calls $halts$ with argument $\overline{\textbf{app}(pr, \textbf{str}(w))}$.

Proof (cont.).

- If *halts* returns **const**(true) (so we know that $\textbf{app}(pr, \textbf{str}(w))$ halts), then *acc*

Proof (cont.).

- If *halts* returns **const**(true) (so we know that **app**(pr, **str**(w)) halts), then *acc* applies the interpreter function to **app**(pr, **str**(w)), using it to evaluate **app**(pr, **str**(w)). If the interpreter returns **const**(true), then *acc* returns **const**(true). Otherwise, the interpreter returns some other value (maybe an error indication), and *acc* returns **const**(false).

Proof (cont.).

- If *halts* returns **const**(true) (so we know that **app**(pr, **str**(w)) halts), then *acc* applies the interpreter function to **app**(pr, **str**(w)), using it to evaluate **app**(pr, **str**(w)). If the interpreter returns **const**(true), then *acc* returns **const**(true). Otherwise, the interpreter returns some other value (maybe an error indication), and *acc* returns **const**(false).
- Otherwise, *halts* returns **const**(false) (so we know that **app**(pr, **str**(w)) does not halt), in which case *acc* returns **const**(false).

Proof (cont.).

- If *halts* returns **const**(true) (so we know that **app**(pr, **str**(w)) halts), then *acc* applies the interpreter function to **app**(pr, **str**(w)), using it to evaluate **app**(pr, **str**(w)). If the interpreter returns **const**(true), then *acc* returns **const**(true). Otherwise, the interpreter returns some other value (maybe an error indication), and *acc* returns **const**(false).
- Otherwise, *halts* returns **const**(false) (so we know that **app**(pr, **str**(w)) does not halt), in which case *acc* returns **const**(false).

Now, we prove that *acc* is a string predicate program testing whether a string is in L_a.

Proof (cont.).

- Suppose $w \in L_a$. Thus $w \in L(pr)$, where pr is the unique closed program described by w. Hence $\mathbf{eval}(\mathbf{app}(pr, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\mathsf{true}))$. It is easy to show that $\mathbf{eval}(\mathbf{app}(acc, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\mathsf{true}))$.

Proof (cont.).

- Suppose $w \in L_a$. Thus $w \in L(pr)$, where pr is the unique closed program described by w. Hence $\mathbf{eval}(\mathbf{app}(pr, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\mathrm{true}))$. It is easy to show that $\mathbf{eval}(\mathbf{app}(acc, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\mathrm{true}))$.
- Suppose $w \notin L_a$. If $w \notin \Sigma^*$, or $w \in \Sigma^*$ but w does not describe a program, or w describes a program that isn't closed, then $\mathbf{eval}(\mathbf{app}(acc, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\mathrm{false}))$.

Proof (cont.).

- Suppose $w \in L_a$. Thus $w \in L(pr)$, where pr is the unique closed program described by w. Hence $\mathbf{eval}(\mathbf{app}(pr, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\mathrm{true}))$. It is easy to show that $\mathbf{eval}(\mathbf{app}(acc, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\mathrm{true}))$.
- Suppose $w \notin L_a$. If $w \notin \Sigma^*$, or $w \in \Sigma^*$ but w does not describe a program, or w describes a program that isn't closed, then $\mathbf{eval}(\mathbf{app}(acc, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\mathrm{false}))$. So, suppose w describes the closed program pr. Then $w \notin L(pr)$, i.e., $\mathbf{eval}(\mathbf{app}(pr, \mathbf{str}(w))) \neq \mathbf{norm}(\mathbf{const}(\mathrm{true}))$. It is easy to show that $\mathbf{eval}(\mathbf{app}(acc, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\mathrm{false}))$.

Proof (cont.).

- Suppose $w \in L_a$. Thus $w \in L(pr)$, where pr is the unique closed program described by w. Hence $\mathbf{eval}(\mathbf{app}(pr, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\mathsf{true}))$. It is easy to show that $\mathbf{eval}(\mathbf{app}(acc, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\mathsf{true}))$.
- Suppose $w \notin L_a$. If $w \notin \Sigma^*$, or $w \in \Sigma^*$ but w does not describe a program, or w describes a program that isn't closed, then $\mathbf{eval}(\mathbf{app}(acc, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\mathsf{false}))$. So, suppose w describes the closed program pr. Then $w \notin L(pr)$, i.e., $\mathbf{eval}(\mathbf{app}(pr, \mathbf{str}(w))) \neq \mathbf{norm}(\mathbf{const}(\mathsf{true}))$. It is easy to show that $\mathbf{eval}(\mathbf{app}(acc, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\mathsf{false}))$.

Thus L_a is recursive—contradiction. Thus there is no such *halt*.
$\square$

We say that a value pr *halts on* a value pr' iff $\mathbf{eval}(\mathbf{app}(pr, pr')) \neq \mathbf{nonterm}$.

Corollary 5.3.8 (Undecidability of the Halting Problem)
There is no value $haltsOn$ such that, for all values pr and pr':

- *if pr halts on pr', then* $\mathbf{eval}(\mathbf{app}(haltsOn, \mathbf{pair}(\overline{pr}, \overline{pr'}))) = \mathbf{norm}(\mathbf{const}(\mathrm{true}))$*; and*
- *If pr does not halt on pr', then* $\mathbf{eval}(\mathbf{app}(haltsOn, \mathbf{pair}(\overline{pr}, \overline{pr'}))) = \mathbf{norm}(\mathbf{const}(\mathrm{false}))$*.*

We say that a value pr *halts on* a value pr' iff $\mathbf{eval}(\mathbf{app}(pr, pr')) \neq \mathbf{nonterm}$.

Corollary 5.3.8 (Undecidability of the Halting Problem)
There is no value $haltsOn$ such that, for all values pr and pr':

- *if pr halts on pr', then* $\mathbf{eval}(\mathbf{app}(haltsOn, \mathbf{pair}(\overline{pr}, \overline{pr'}))) = \mathbf{norm}(\mathbf{const}(\mathrm{true}))$*; and*
- *If pr does not halt on pr', then* $\mathbf{eval}(\mathbf{app}(haltsOn, \mathbf{pair}(\overline{pr}, \overline{pr'}))) = \mathbf{norm}(\mathbf{const}(\mathrm{false}))$*.*

Proof. Suppose, toward a contradiction, that such a $haltsOn$ exists. Let $halts$ be the value that takes in a value $\overline{pr}$ representing a closed program pr, and then returns the result of calling $haltsOn$ with $\mathbf{pair}(\overline{\mathbf{lam}(\mathrm{x}, pr)}, \overline{\mathbf{const}(\mathrm{nil})})$. Then this value satisfies the property of Theorem 5.3.7—contradiction. Thus such a $haltsOn$ does not exist. $\square$

Other Undecidable Problems

Here are two other undecidable problems:

- Determining whether two grammars generate the same language. (In contrast, we gave an algorithm for checking whether two FAs are equivalent, and this algorithm can be implemented as a program.)
- Determining whether a grammar is ambiguous.