

Section 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.

Copyright © 2003–4 Alley Stoughton

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled “GNU Free Documentation License”.

The \LaTeX source of these slides, the associated book, and the distribution of the Forlan toolset are available on the WWW at <http://people.cis.ksu.edu/~stough/forlan/>.

(5.3) Diagonalization

To find a non-r.e. language, we can use a technique called “diagonalization”. Remember that the alphabet **Syn** consists of the digits, the lowercase and uppercase letters, and the symbols $\langle \text{space} \rangle$, $\langle \text{newline} \rangle$, $\langle \text{openPar} \rangle$ and $\langle \text{closPar} \rangle$. Furthermore $\mathbf{Prog} \subseteq \mathbf{Syn}^*$.

Consider the infinite table in which both the rows and the columns are indexed by the elements of \mathbf{Syn}^* , listed in some order w_1, w_2, \dots , and where a cell (w_n, w_m) contains 1 iff $\mathbf{run}_1(w_n, w_m) = \mathbf{true}$, and contains 0 iff $\mathbf{run}_1(w_n, w_m) \neq \mathbf{true}$. Each recursively enumerable language is $L(w_n)$ for some n .

(5.3) Diagonalization (Cont.)

Here is how part of this table might look, where w_i , w_j and w_k are sample elements of \mathbf{Syn}^* :

	...	w_i	...	w_j	...	w_k	...
⋮							
w_i		1		1		0	
⋮							
w_j		0		0		1	
⋮							
w_k		0		1		1	
⋮							

(5.3) Diagonalization (Cont.)

Because of the table's data, we have that $\text{run}_1(w_i, w_i) = \text{true}$ and $\text{run}_1(w_i, w_j) \neq \text{true}$.

To define a non-r.e. 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 $\text{run}_1(w_n, w_n) \neq \text{true}$. Thus, if w_n is a program, we will have that w_n is in our language exactly when w_n doesn't accept w_n . This will ensure that $L(w_n)$ (if it is defined) is not our language.

With our example table:

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

(5.3) Diagonalization (Cont.)

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

$$L_d = \{ w \in \mathbf{Syn}^* \mid \mathbf{run}_1(w, w) \neq \mathbf{true} \},$$
$$L_a = \{ w \in \mathbf{Syn}^* \mid \mathbf{run}_1(w, w) = \mathbf{true} \}.$$

Thus $L_d = \mathbf{Syn}^* - L_a$. Because of the way \mathbf{run}_1 is defined, we have that every element of L_a is a program whose principal function has a single argument.

(5.3) Diagonalization (Cont.)

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 program P such that $L_d = L(P)$. There are two cases to consider.

Suppose $P \in L_d$. Then $\text{run}_1(P, P) \neq \text{true}$, i.e., P is not accepted by P . But then $P \notin L(P) = L_d$ —contradiction.

Suppose $P \notin L_d$. Since $P \in \text{Syn}^*$, we have that $\text{run}_1(P, P) = \text{true}$, i.e., P is accepted by P . But then $P \in L(P) = L_d$ —contradiction.

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

(5.3) Diagonalization (Cont.)

Theorem 5.3.2

L_a is recursively enumerable.

Proof. Let P be the program that, when given a string w , uses our interpreter function to simulate the execution of w on input w , returning **true** if the interpreter returns **true**, returning **false** if the interpreter returns **false** or **error**, and running forever, if the interpreter runs forever.

We can check that, for all $w \in \mathbf{Str}$,

$w \in L_a = \{ w \in \mathbf{Syn}^* \mid \mathbf{run}_1(w, w) = \mathbf{true} \}$ iff $\mathbf{run}_1(P, w) = \mathbf{true}$.

Thus L_a is recursively enumerable. \square

(5.3) Diagonalization (Cont.)

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. $L_a \subseteq \mathbf{Syn}^*$ is recursively enumerable, but $\mathbf{Syn}^* - L_a = L_d$ is not recursively enumerable. \square

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. \square

(5.3) Diagonalization (Cont.)

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 $L_a \subseteq \mathbf{Syn}^*$, we have that $L_d = \mathbf{Syn}^* - L_a$ is recursive—contradiction. Thus L_a is not recursive. \square

(5.3) Relationship Between the Context-Free, Recursive and Recursive Enumerable Languages

Since $L_a \in \mathbf{RELang}$ and $L_a \notin \mathbf{RecLang}$, we have:

Theorem 5.3.6

The recursive languages are a proper subset of the recursively enumerable languages: $\mathbf{RecLang} \subsetneq \mathbf{RELang}$.

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

$$\mathbf{RegLang} \subsetneq \mathbf{CFLang} \subsetneq \mathbf{RecLang} \subsetneq \mathbf{RELang} \subsetneq \mathbf{Lang}.$$

(5.3) Undecidability of the Halting Problem

We say that a program P halts on a string w iff $\text{run}_1(P, w) \neq \text{nonterm}$.

Theorem 5.3.7

There is no program H such that, for all programs P and strings w :

- If P halts on w , then $\text{run}_2(H, (P, w)) = \text{true}$;
- If P does not halt on w , then $\text{run}_2(H, (P, w)) = \text{false}$.

Proof. Suppose, toward a contradiction, that such an H does exist. We use H to construct a program Q that behaves as follows when run on a string w . If w is not a program whose principal function has a single argument, then it returns **false**. Otherwise, we call the principal function of H with inputs (w, w) .

(5.3) Undecidability of Halting Problem (Cont.)

Proof (cont.). If H returns **true**, then Q uses our interpreter function to run w with input w . Since w halts on w , we know that this interpretation will terminate. If it terminates with value **true**, then Q returns **true**. Otherwise, it returns **false**.

Otherwise, H returns **false**. Then Q returns **false**.

We can check that, for all $w \in \mathbf{Str}$, Q tests whether $w \in L_a = \{ w \in \mathbf{Syn}^* \mid \mathbf{run}_1(w, w) = \mathbf{true} \}$. Thus L_a is recursive—contradiction. Thus no such H exists. \square

(5.3) 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.