5.2: Closure Properties of Recursive and Recursively Enumerable Languages

In this section, we will see that the recursive and recursively enumerable languages are closed under union, concatenation, closure and intersection.

The recursive languages are also closed under set difference and complementation.

In the next section, we will see that the recursively enumerable languages are not closed under complementation or set difference.

On the other hand, we will see in this section that, if a language and its complement are both r.e., then the language is recursive.

Closure Properties of Recursive Languages

Theorem 5.2.1

If L, L_1 and L_2 are recursive languages, then so are $L_1 \cup L_2$, $L_1 L_2$, L^*, $L_1 \cap L_2$ and $L_1 - L_2$.

Proof.

Closure Properties of Recursive Languages

Theorem 5.2.1
If L, L_1 and L_2 are recursive languages, then so are $L_1 \cup L_2$, $L_1 L_2$, L^*, $L_1 \cap L_2$ and $L_1 - L_2$.

Proof. Let's consider the concatenation case as an example. Since L_1 and L_2 are recursive languages, there are string predicate programs pr_1 and pr_2 that test whether strings are in L_1 and L_2, respectively.

Proof (cont.). We write a program pr with form

$$\begin{array}{l}\textbf{lam}(\mathsf{w},\\ \quad \textbf{letSimp}(\mathsf{f1},\\ \qquad pr_1,\\ \qquad \textbf{letSimp}(\mathsf{f2},\\ \qquad\quad pr_2,\\ \qquad\quad \cdots))),\end{array}$$

which tests whether its input is an element of $L_1 L_2$.

Proof (cont.). The elided part of *pr* generates all of the pairs of strings (x_1, x_2) such that $x_1 x_2$ is equal to the value of w. Then it works though these pairs, one by one.

- Given such a pair (x_1, x_2), *pr* calls f1 with x_1 to check whether $x_1 \in L_1$. If the answer is **const(false)**, then it goes on to the next pair.
- Otherwise, it calls f2 with x_2 to check whether $x_2 \in L_2$. If the answer is **const(false)**, then it goes on to the next pair. Otherwise, it returns **const(true)**.
- If *pr* runs out of pairs to check, then it returns **const(false)**.

We can check that *pr* is a string predicate program testing whether $w \in L_1 L_2$. Thus $L_1 L_2$ is recursive. □

Closure Properties of Rec. Lan.

Corollary 5.2.2
If Σ is an alphabet and $L \subseteq \Sigma^*$ is recursive, then so is $\Sigma^* - L$.

Proof. Follows from Theorem 5.2.1, since Σ^* is recursive. □

Theorem 5.2.2
If L, L_1 and L_2 are recursively enumerable languages, then so are $L_1 \cup L_2$, $L_1 L_2$, L^* and $L_1 \cap L_2$.

Proof. We consider the concatenation case as an example. Since L_1 and L_2 are recursively enumerable, there are closed programs pr_1 and pr_2 such that, for all $w \in \mathbf{Str}$, $w \in L_1$ iff $\mathbf{eval}(\mathbf{app}(pr_1, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\mathbf{true}))$, and for all $w \in \mathbf{Str}$, $w \in L_2$ iff $\mathbf{eval}(\mathbf{app}(pr_2, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\mathbf{true}))$. (Remember that pr_1 and pr_2 may fail to terminate on some inputs.)

Theorem 5.2.2
If L, L_1 and L_2 are recursively enumerable languages, then so are $L_1 \cup L_2$, $L_1 L_2$, L^* and $L_1 \cap L_2$.

Proof. We consider the concatenation case as an example. Since L_1 and L_2 are recursively enumerable, there are closed programs pr_1 and pr_2 such that, for all $w \in \mathbf{Str}$, $w \in L_1$ iff $\mathbf{eval}(\mathbf{app}(pr_1, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\mathbf{true}))$, and for all $w \in \mathbf{Str}$, $w \in L_2$ iff $\mathbf{eval}(\mathbf{app}(pr_2, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\mathbf{true}))$. (Remember that pr_1 and pr_2 may fail to terminate on some inputs.)

To show that $L_1 L_2$ is recursively enumerable, we will construct a closed program pr such that, for all $w \in \mathbf{Str}$, $w \in L_1 L_2$ iff $\mathbf{eval}(\mathbf{app}(pr, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\mathbf{true}))$.

Proof (cont.). When pr is called with $\mathbf{str}(w)$, for some $w \in \mathbf{Str}$, it behaves as follows. First, it generates all the pairs of strings (x_1, x_2) such that $w = x_1 x_2$. Let these pairs be $(x_{1,1}, x_{2,1}), \ldots, (x_{1,n}, x_{2,n})$.

Proof (cont.). When pr is called with $\mathbf{str}(w)$, for some $w \in \mathbf{Str}$, it behaves as follows. First, it generates all the pairs of strings (x_1, x_2) such that $w = x_1 x_2$. Let these pairs be $(x_{1,1}, x_{2,1}), \ldots, (x_{1,n}, x_{2,n})$. Now, pr uses our *incremental* interpretation function to run a fixed number of steps of $\mathbf{app}(pr_1, \mathbf{str}(x_{1,i}))$ and $\mathbf{app}(pr_2, \mathbf{str}(x_{2,i}))$ (working with $\overline{\mathbf{app}(pr_1, \mathbf{str}(x_{1,i}))}$ and $\overline{\mathbf{app}(pr_2, \mathbf{str}(x_{2,i}))}$), for all $i \in [1 : n]$, and then repeat this over and over again.

Proof (cont.). When pr is called with $\mathbf{str}(w)$, for some $w \in \mathbf{Str}$, it behaves as follows. First, it generates all the pairs of strings (x_1, x_2) such that $w = x_1 x_2$. Let these pairs be $(x_{1,1}, x_{2,1}), \ldots, (x_{1,n}, x_{2,n})$. Now, pr uses our *incremental* interpretation function to run a fixed number of steps of $\mathbf{app}(pr_1, \mathbf{str}(x_{1,i}))$ and $\mathbf{app}(pr_2, \mathbf{str}(x_{2,i}))$ (working with $\overline{\mathbf{app}(pr_1, \mathbf{str}(x_{1,i}))}$ and $\overline{\mathbf{app}(pr_2, \mathbf{str}(x_{2,i}))}$), for all $i \in [1 : n]$, and then repeat this over and over again.

- If, at some stage, the incremental interpretation of $\mathbf{app}(pr_1, \mathbf{str}(x_{1,i}))$ returns $\overline{\mathbf{const}(\mathbf{true})}$, then $x_{1,i}$ is marked as being in L_1.
- If, at some stage, the incremental interpretation of $\mathbf{app}(pr_2, \mathbf{str}(x_{2,i}))$ returns $\overline{\mathbf{const}(\mathbf{true})}$, then the $x_{2,i}$ is marked as being in L_2.

Proof (cont.).

- If, at some stage, the incremental interpretation of $\overline{\mathbf{app}(pr_1, \mathbf{str}(x_{1,i}))}$ returns something other than $\overline{\mathbf{const}(\mathbf{true})}$, then the i'th pair is marked as discarded.
- If, at some stage, the incremental interpretation of $\overline{\mathbf{app}(pr_2, \mathbf{str}(x_{2,i}))}$ returns something other than $\overline{\mathbf{const}(\mathbf{true})}$, then the i'th pair is marked as discarded.
- If, at some stage, $x_{1,i}$ is marked as in L_1 and $x_{2,i}$ is marked as in L_2, then Q returns $\mathbf{const}(\mathbf{true})$.
- If, at some stage, there are no remaining pairs, then pr returns $\mathbf{const}(\mathbf{false})$.

□

Closure Properties of R.E. Lan.

Theorem 5.2.3

If Σ is an alphabet, $L \subseteq \Sigma^*$ is a recursively enumerable language, and $\Sigma^* - L$ is recursively enumerable, then L is

Closure Properties of R.E. Lan.

Theorem 5.2.3

If Σ is an alphabet, $L \subseteq \Sigma^*$ is a recursively enumerable language, and $\Sigma^* - L$ is recursively enumerable, then L is recursive.

Closure Properties of R.E. Lan.

Theorem 5.2.3
If Σ is an alphabet, $L \subseteq \Sigma^*$ is a recursively enumerable language, and $\Sigma^* - L$ is recursively enumerable, then L is recursive.

Proof. Since L and $\Sigma^* - L$ are recursively enumerable languages, there are closed programs pr_1 and pr_2 such that, for all $w \in \mathbf{Str}$, $w \in L$ iff $\mathbf{eval}(\mathbf{app}(pr_1, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\mathbf{true}))$, and for all $w \in \mathbf{Str}$, $w \in \Sigma^* - L$ iff $\mathbf{eval}(\mathbf{app}(pr_2, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\mathbf{true}))$.

Theorem 5.2.3
If Σ is an alphabet, $L \subseteq \Sigma^*$ is a recursively enumerable language, and $\Sigma^* - L$ is recursively enumerable, then L is recursive.

Proof. Since L and $\Sigma^* - L$ are recursively enumerable languages, there are closed programs pr_1 and pr_2 such that, for all $w \in \mathbf{Str}$, $w \in L$ iff $\mathbf{eval}(\mathbf{app}(pr_1, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\mathbf{true}))$, and for all $w \in \mathbf{Str}$, $w \in \Sigma^* - L$ iff $\mathbf{eval}(\mathbf{app}(pr_2, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\mathbf{true}))$.

We construct a string predicate program pr that tests whether its input is in L. Given $\mathbf{str}(w)$, for $w \in \mathbf{Str}$, pr proceeds as follows. If $w \notin \Sigma^*$, then pr returns $\mathbf{const}(\mathbf{false})$. Otherwise,

Closure Properties of R.E. Lan.

Theorem 5.2.3
If Σ is an alphabet, $L \subseteq \Sigma^*$ is a recursively enumerable language, and $\Sigma^* - L$ is recursively enumerable, then L is recursive.

Proof. Since L and $\Sigma^* - L$ are recursively enumerable languages, there are closed programs pr_1 and pr_2 such that, for all $w \in \mathbf{Str}$, $w \in L$ iff $\mathbf{eval}(\mathbf{app}(pr_1, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\mathbf{true}))$, and for all $w \in \mathbf{Str}$, $w \in \Sigma^* - L$ iff $\mathbf{eval}(\mathbf{app}(pr_2, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\mathbf{true}))$.

We construct a string predicate program pr that tests whether its input is in L. Given $\mathbf{str}(w)$, for $w \in \mathbf{Str}$, pr proceeds as follows. If $w \notin \Sigma^*$, then pr returns $\mathbf{const}(\mathbf{false})$. Otherwise, pr alternates between incrementally interpreting $\mathbf{app}(pr_1, \mathbf{str}(w))$ (working with $\overline{\mathbf{app}(pr_1, \mathbf{str}(w))}$) and $\mathbf{app}(pr_2, \mathbf{str}(w))$ (working with $\overline{\mathbf{app}(pr_2, \mathbf{str}(w))}$).

Proof (cont.).

- If, at some stage, the first incremental interpretation returns $\overline{\textbf{const}(\textbf{true})}$, then *pr* returns $\textbf{const}(\textbf{true})$.
- If, at some stage, the second incremental interpretation returns $\overline{\textbf{const}(\textbf{true})}$, then *pr* returns
- If, at some stage, the first incremental interpretation returns anything other than $\overline{\textbf{const}(\textbf{true})}$, then *pr* returns

- If, at some stage, the second incremental interpretation returns anything other than $\overline{\textbf{const}(\textbf{true})}$, then *pr* returns

□

Proof (cont.).

- If, at some stage, the first incremental interpretation returns $\overline{\textbf{const}(\textbf{true})}$, then *pr* returns $\textbf{const}(\textbf{true})$.
- If, at some stage, the second incremental interpretation returns $\overline{\textbf{const}(\textbf{true})}$, then *pr* returns $\textbf{const}(\textbf{false})$.
- If, at some stage, the first incremental interpretation returns anything other than $\overline{\textbf{const}(\textbf{true})}$, then *pr* returns

- If, at some stage, the second incremental interpretation returns anything other than $\overline{\textbf{const}(\textbf{true})}$, then *pr* returns

□

Proof (cont.).

- If, at some stage, the first incremental interpretation returns $\overline{\textbf{const}(\textbf{true})}$, then *pr* returns $\textbf{const}(\textbf{true})$.
- If, at some stage, the second incremental interpretation returns $\overline{\textbf{const}(\textbf{true})}$, then *pr* returns $\textbf{const}(\textbf{false})$.
- If, at some stage, the first incremental interpretation returns anything other than $\overline{\textbf{const}(\textbf{true})}$, then *pr* returns $\textbf{const}(\textbf{false})$.
- If, at some stage, the second incremental interpretation returns anything other than $\overline{\textbf{const}(\textbf{true})}$, then *pr* returns

□

Proof (cont.).

- If, at some stage, the first incremental interpretation returns $\overline{\textbf{const}(\textbf{true})}$, then *pr* returns $\textbf{const}(\textbf{true})$.
- If, at some stage, the second incremental interpretation returns $\overline{\textbf{const}(\textbf{true})}$, then *pr* returns $\textbf{const}(\textbf{false})$.
- If, at some stage, the first incremental interpretation returns anything other than $\overline{\textbf{const}(\textbf{true})}$, then *pr* returns $\textbf{const}(\textbf{false})$.
- If, at some stage, the second incremental interpretation returns anything other than $\overline{\textbf{const}(\textbf{true})}$, then *pr* returns $\textbf{const}(\textbf{true})$.

□