Question 1: A Variation of the Language $\mathcal{L}$

Let us consider a new language $\mathcal{L}'$. This language is identical to $\mathcal{L}$, except that instead of the instruction types

\[
V \leftarrow V \\
V \leftarrow V + 1 \\
V \leftarrow V - 1 \\
\text{IF } V \neq 0 \text{ GOTO L}
\]

we now have the instruction types

\[
V \leftarrow V \\
V \leftarrow V + 3 \\
V \leftarrow V - 2 \quad (\text{again, if } V \text{ is 0 or 1, it becomes 0 after this operation}) \\
\text{IF } V = 1 \text{ GOTO L}
\]

(a) Show that the language $\mathcal{L}'$ is at least as powerful as $\mathcal{L}$, that is, can compute all functions that $\mathcal{L}$ can compute. **Hint:** You need to show for each instruction type in $\mathcal{L}$ how it can be simulated in $\mathcal{L}'$, i.e., how we could translate it into $\mathcal{L}'$ code that has the same effect.

(b) Write down the entire $\mathcal{L}'$ code for the new universal programs $U'_n$ that can execute the code of any $\mathcal{L}'$ program. Of course you can use macros, and you can reuse most of the code that we wrote for programming $U_n$ (see slides and textbook).
(a) We can translate any $L$ program into an equivalent $L'$ program using the following rules for each type of instruction in $L$:

(I) $V \leftarrow V$

This instruction type simply remains unchanged, because we have the same type in $L'$.

(II) $V \leftarrow V + 1$

This instruction type can be translated into the following two $L'$ instructions:

$V \leftarrow V + 3$
$V \leftarrow V - 2$

(III) $V \leftarrow V - 1$

This instruction type can be translated into the following three $L'$ instructions:

$V \leftarrow V + 3$
$V \leftarrow V - 2$
$V \leftarrow V - 2$

(IV) IF $V \neq 0$ GOTO $L$

This instruction type can be translated into the following 13 $L'$ instructions:

$V \leftarrow V + 3$ // Increment $V$
$V \leftarrow V - 2$
IF $V = 1$ GOTO $L_2$ // If $V$ was originally 0 (no branch), go to $L_2$
$V \leftarrow V + 3$ // Decrement $V$ to restore its original value
$V \leftarrow V - 2$
$V \leftarrow V - 2$
$V_2 \leftarrow V_2 - 2$ // Make sure $V_2$ is 0 (in case we are in a loop)
$V_2 \leftarrow V_2 + 3$ // Set $V_2$ to 1 to enable the (unconditional) branch
$V_2 \leftarrow V_2 - 2$
IF $V_2 = 1$ GOTO $L$ // Go to $L$
$[L_2]$ $V \leftarrow V + 3$ // Decrement $V$ to restore its original value
$V \leftarrow V - 2$
$V \leftarrow V - 2$

Here, $L_2$ and $V_2$ have to be chosen such that they are not used anywhere else in the program.

Using the above rules, we can translate any $L$ program into an $L'$ program that computes the same function. Therefore, we have shown that the language $L'$ is at least as powerful as the
language $\mathcal{L}$.

(b) We only need to change those parts of the universal programs that relate to the interpretation of instruction types $V \leftarrow V + 1$, $V \leftarrow V - 1$, and IF $V \neq 0$ GOTO L. Since we can now translate any $\mathcal{L}$ program into an $\mathcal{L}'$ program, we can still use the macros we defined in $\mathcal{L}$. In the following solution, the modified instructions are shown in **bold font**:

<table>
<thead>
<tr>
<th>Original Program</th>
<th>Modified Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Z \leftarrow X_{n+1} + 1$</td>
<td>$Z \leftarrow X_{n+1} + 1$</td>
</tr>
<tr>
<td>$S \leftarrow \prod_{i=1}^{n} (p_2)_i^{X_i}$</td>
<td>$S \leftarrow \prod_{i=1}^{n} (p_2)_i^{X_i}$</td>
</tr>
<tr>
<td>$K \leftarrow 1$</td>
<td>$K \leftarrow 1$</td>
</tr>
<tr>
<td>[C] IF $K = \text{Lt}(Z) + 1 \lor K = 0$ GOTO F</td>
<td>[C] IF $K = \text{Lt}(Z) + 1 \lor K = 0$ GOTO F</td>
</tr>
<tr>
<td>$U \leftarrow r((Z)_k)$</td>
<td>$U \leftarrow r((Z)_k)$</td>
</tr>
<tr>
<td>$P \leftarrow p_{r(U)+1}$</td>
<td>$P \leftarrow p_{r(U)+1}$</td>
</tr>
<tr>
<td>IF $l(U) = 0$ GOTO N</td>
<td>IF $l(U) = 0$ GOTO N</td>
</tr>
<tr>
<td>IF $l(U) = 1$ GOTO A</td>
<td>IF $l(U) = 1$ GOTO A</td>
</tr>
<tr>
<td>IF $\sim(P \mid S)$ GOTO N</td>
<td>IF $\sim(P \mid S)$ GOTO N</td>
</tr>
<tr>
<td>IF $l(U) = 2$ GOTO M</td>
<td>IF $l(U) = 2$ GOTO M</td>
</tr>
<tr>
<td>$K \leftarrow \min_{i \leq \text{Lt}(Z)} [l((Z)_i) + 2 = l(U)]$</td>
<td>$K \leftarrow \min_{i \leq \text{Lt}(Z)} [l((Z)_i) + 2 = l(U)]$</td>
</tr>
<tr>
<td>GOTO C</td>
<td>GOTO C</td>
</tr>
<tr>
<td>[M] $S \leftarrow \lfloor S/P \rfloor$</td>
<td>[M] $S \leftarrow \lfloor S/P \rfloor$</td>
</tr>
<tr>
<td>GOTO N</td>
<td>GOTO N</td>
</tr>
<tr>
<td>[A] $S \leftarrow S \cdot P$</td>
<td>[A] $S \leftarrow S \cdot P^3$</td>
</tr>
<tr>
<td>[N] $K \leftarrow K + 1$</td>
<td>[N] $K \leftarrow K + 1$</td>
</tr>
<tr>
<td>GOTO C</td>
<td>GOTO C</td>
</tr>
<tr>
<td>[F] $Y \leftarrow (S)_i$</td>
<td>[F] $Y \leftarrow (S)_i$</td>
</tr>
</tbody>
</table>
Question 2: Some Set Operations

Let A and B be sets. Prove or disprove:

(a) For all sets A and B, if A is recursive and B is r.e., then $A \cup B$ is also r.e.

Because A is recursive, there is a computable predicate $P_A(x)$ such that:

$$A = \{ x \in \mathbb{N} \mid P_A(x) \}$$

Since B is r.e., there must be a partially computable functions $f_B(x)$ such that:

$$B = \{ x \in \mathbb{N} \mid f_B(x) \downarrow \}$$

Then we can write the following program computing a function $f_{A\cup B}(x)$:

```
IF $P_A(X)$ GOTO E
Z ← $f_B(X)$
```

This program will terminate on input $x$ if and only if $x$ is in A or in B, i.e.:

$$A \cup B = \{ x \in \mathbb{N} \mid f_{A\cup B}(x) \downarrow \}$$

Since we have shown that $f_{A\cup B}(x)$ is a partially computable function, $A \cup B$ is r.e.

Alternative solution:

Since A is recursive, A is also r.e.

If A and B are both r.e., then there must be partially computable functions $f_A(x)$ and $f_B(x)$ such that:

$$A = \{ x \in \mathbb{N} \mid f_A(x) \downarrow \}$$
$$B = \{ x \in \mathbb{N} \mid f_B(x) \downarrow \}$$

Because these functions are partially computable, there are programs computing them, and therefore, we can use these functions as macros in our programs.

In order to show that $A \cup B$ is r.e., we have to show that there is a partially computable function $f_{A\cup B}(x)$ such that:

$$A \cup B = \{ x \in \mathbb{N} \mid f_{A\cup B}(x) \downarrow \}$$

We can show that there is such a partially computable function simply by giving a program
that computes it. Writing a program that computes $f_{A \cup B}(x)$ is tricky, because we cannot simply compute $f_A(X)$ and then $f_B(X)$ and check whether either one terminated and then return 1 and otherwise 0. The problem is that if we call $f_A(X)$ and it never terminates, we will be stuck and not be able to check whether $f_B(X)$ terminates.

The solution here is the dovetailing technique that we used before. We use the step counter function to execute one step of function $f_A(X)$, and if it does not reach a terminal state, we execute one step of function $f_B(X)$. If that one does not reach a terminal state, either, we run two steps of $f_A(X)$ and, if necessary, two steps of $f_B(X)$, and so on. Once either of the two functions reaches a terminal state, then our program also terminates, and otherwise it keeps on running and simulating more and more steps for both functions. Here is a program that performs these computations:

\[
\begin{align*}
[A] & \quad Z \leftarrow Z + 1 \\
& \quad \text{IF STP}^{(1)}(X, #(f_A), Z) \text{ GOTO E} \\
& \quad \text{IF STP}^{(1)}(X, #(f_B), Z) \text{ GOTO E} \\
& \quad \text{GOTO A}
\end{align*}
\]

If either $f_A(X)$ or $f_B(X)$ terminates for a given input $X$, then this program will terminate. Otherwise, it will never terminate. This means that the function this program computes has exactly the required properties for $f_{A \cup B}(X)$, and therefore, we have shown that the set $A \cup B$ is r.e.

(b) For all sets $A$ and $B$, if $A$ and $B$ are both recursive, then $A - B$ is also recursive.

If $A$ and $B$ are both recursive, then there must be computable predicates $P_A(x)$ and $P_B(x)$ such that:

\[
A = \{ x \in \mathbb{N} \mid P_A(x) \} \\
B = \{ x \in \mathbb{N} \mid P_B(x) \}
\]

Because these predicates are computable, there are programs computing them, and we can rely on them to terminate within a finite amount of steps and return 0 (or false, meaning their input is not in the given set) or 1 (or true, meaning their input is in the given set). Again, we can use these functions as macros in our programs.

In order to show that $A - B$ is recursive, we have to show that there is a computable predicate $P_{A-B}(x)$ such that:

\[
A - B = \{ x \in \mathbb{N} \mid P_{A-B}(x) \}
\]

We can show that there is such a partially computable function simply by giving a program that computes it. The following program computes $P_{A-B}(x)$:
IF \( P_A(X) = 0 \lor P_B(X) = 1 \) GOTO E
\[ Y \leftarrow 1 \]

If \( P_A(X) \) returns true and \( P_B(X) \) returns false, then this program will return true. In all other cases, it will return 1, and it can never enter an infinite loop. This means that the function this program computes is \( P_{A-B}(X) \), and therefore, we have shown that the set \( A - B \) is recursive.

Alternatively, we could just say that \( A - B \) can be computed as:
\[ A - B = \{ x \in N \mid P_A(x) \land \neg P_B(x) \} \]

Since \( P_A(x) \) and \( P_B(x) \) are both in the PRC class of computable functions and \( \land \) and \( \neg \) are primitive recursive functions, \( A - B \) must be recursive.

(c) If \( A \subset B \) and \( A \) is r.e., then \( B \) is r.e.

This is false. We can disprove it by providing a counterexample: As we have shown in class, the empty set is r.e. We also showed that the set \( \neg K \) is not r.e.

Now let \( A = \emptyset \) and \( B = \neg K \). Then \( A \subset B \) and \( A \) is r.e., but \( B \) is not r.e. End of proof.

(d) If \( A \cup B \) is r.e., then both \( A \) and \( B \) are r.e.

Same idea as above: Let \( A = \neg K \) and \( B = N \). Then \( A \cup B \) is r.e., but \( A \) is not r.e. This counterexample shows that the statement is false.

(e) If \( A \) is r.e, then \( \neg A \) is also r.e.

False. Counterexample: Let \( A = K \). Then \( A \) is r.e. but \( \neg A = \neg K \) is not r.e.

Question 3 (Bonus Question): What about These Sets?

For each of the following sets, determine whether it is recursive, r.e., or neither. Prove your answer. Of course, if you prove that a set is recursive, it is clear that it is also r.e., and you do not have to prove that.

(a) \( A = \{ x \in N \mid x^2 > 1000 \} \)

\( A \) is recursive, because the following program computes the predicate \( P_{A}(x) \) that tests for membership in \( A \):
Z ← X⋅X
IF Z ≤ 1000 GOTO E
Y ← 1

It would also suffice to state that \((x^2 > 1000)\) is a computable predicate, because multiplication and “greater than” are primitive recursive, and therefore, A is also recursive.

(b) \(B = \{ x ∈ \mathbb{N} | x \) is the number of a program that computes the function \(f(n) = 0 \}\}

According to Rice’s theorem, we cannot algorithmically determine whether a program computes a given function. Therefore, there is neither a computable predicate \(P_B(x)\) that decides about membership in B nor a partially computable function \(f_B(x)\) that terminates whenever \(x\) is in B. Consequently, B is neither recursive nor r.e.

(c) \(C = \{ x ∈ \mathbb{N} | x \) is the number of a program that terminates on input 12 after 50 or more steps\}

Based on the halting problem we know that there is no computable function that can determine whether a given program is ever going to halt, and therefore, C is not recursive. The condition of “50 or more” steps does not help here, because it does not impose an upper limit on the number of computational steps.

However, this definition still meets the requirements for being r.e. We can show that by writing the following program that computes the function \(f_C(x)\):

\[
\begin{align*}
Z & ← 49 \\
[A] & Z ← Z + 1 \\
& IF STP^{(1)}(12, X, Z) GOTO E \\
& GOTO A
\end{align*}
\]

Then we have:

\(C = \{ x ∈ \mathbb{N} | f_C(x) \downarrow \}\)

Therefore, C is r.e.

(d) \(D = \{ x ∈ \mathbb{N} | x \) is the number of a program whose output is defined for at least 3 different inputs\}

D certainly cannot be recursive, because we could never determine within a finite amount of computational steps that a given program is not defined on any input (or we could argue based on Rice’s theorem).

If D is r.e., we would need to show that there is a partially computable function \(f_D(x)\) that
terminates if and only if x is in D. Writing a program to compute \( f_D(x) \) is difficult for reasons similar to the issue in Question 2(a). If we call \( f_D(0) \) to test whether it terminates and it does not, we will not be able to ever call \( f_D(1) \) or \( f_D(2) \), etc., because we are stuck in computing \( f_D(0) \). And we need to check for all possible inputs x whether \( f_D(x) \) terminates.

As in 2(a), we can use the step-counter function to help us out. Here, however, we do not just have to consider two cases but infinitely many. We have to run every possible input for any number of steps in order to be sure to find an input that \( f_D(x) \) halts on if such an input exists.

In other words, we have to go through all combinations of input and number of steps to be tested, with neither of these two numbers having an upper limit. That reminds us of a trick that we used earlier in the course: If we use the pairing function \( z = <l(z), r(z)> \) and let z run from 0 to infinity, then \( l(z) \) and \( r(z) \) will assume all possible combinations of numbers, with no upper limit to either of them. We could let the program in question (with number x) run for \( l(z) \) steps on input \( r(z) \), using the step-counter function. If it terminates, we check whether it already terminated for the same input before, and if not, we increase a counter. Then we increment z and obtain the next values of \( l(z) \) and \( r(z) \) for testing. This way we will eventually test for any possible input and number of steps whether the program with number x will terminate. Once our counter reaches the number 3, indicating that we found three distinct inputs for which the program halts, the main program will halt. Therefore, the main program will compute \( f_D(x) \). Below is a sample program, using variable Z3 as the counter and variables Z4 and Z5 to remember previous inputs for which the program with number X terminated:

\[
\begin{align*}
[A] & \quad Z \leftarrow Z + 1 \\
& \quad Z_2 \leftarrow \text{STP}(1)(r(Z), X, l(Z)) \\
& \quad \text{IF } Z_2 = 0 \text{ GOTO A} \\
& \quad \text{IF } Z_3 > 0 \text{ GOTO B} \\
& \quad Z_4 \leftarrow r(Z) \\
& \quad Z_5 \leftarrow 1 \\
& \quad \text{GOTO A} \\
[B] & \quad \text{IF } Z_3 > 1 \text{ GOTO C} \\
& \quad \text{IF } r(Z) = Z_4 \text{ GOTO A} \\
& \quad Z_5 \leftarrow r(Z) \\
& \quad Z_3 \leftarrow 2 \\
& \quad \text{GOTO A} \\
[C] & \quad \text{IF } r(Z) = Z_4 \lor r(Z) = Z_5 \text{ GOTO A}
\end{align*}
\]

Consequently, we have shown that \( f_D(x) \) is partially computable, and therefore, D is r.e.