Question 1: A Variation of the Language $L$

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

\[
\begin{align*}
V &\leftarrow V \\
V &\leftarrow V + 1 \\
V &\leftarrow V - 1 \\
\text{IF } V \neq 0 &\text{ GOTO L}
\end{align*}
\]

we now have the instruction types

\[
\begin{align*}
V &\leftarrow V \\
V &\leftarrow V + 2 \\
V &\leftarrow V - 1 \\
\text{IF } V = 0 &\text{ GOTO L}
\end{align*}
\]

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

(b) Write down the entire $L'$ code for the new universal programs $U'_n$ that can execute the code of any $L'$ 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:

$V \leftarrow V + 2$

$V \leftarrow V - 1$

(III) $V \leftarrow V - 1$

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

(IV) $\text{IF } V \neq 0 \text{ GOTO } L$

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

$\text{IF } V = 0 \text{ GOTO } L_2$

$\text{IF } V_2 = 0 \text{ GOTO } L$

$L_2$ ...$

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 $L'$.

(b) We only need to change those parts of the universal programs that relate to the interpretation of instruction types $V \leftarrow V + 1$ and $\text{IF } V \neq 0 \text{ GOTO } L$. In the following solution (see next page), the modified instructions are shown in bold font:
Question 2: Some Set Operations

Let A and B be sets. Prove or disprove:

(a) (Bonus Question) For all sets A and B, if A and B are both r.e., then $A \cup B$ 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:

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

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:

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

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:

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

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 \( A \cup B \) 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 \( A \cup B \) is recursive, we have to show that there is a computable predicate \( P_{A \cup B}(x) \) such that:

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

We can show that there is such a partially computable function simply by giving a program that computes it. The following program computes \( P_{A \cup B}(x) \):

\[
\text{IF } P_A(X) = 0 \land P_B(X) = 0 \text{ GOTO E} \\
Y \leftarrow 1
\]
If both $P_A(X)$ and $P_B(X)$ return 0, then this program will also return 0. 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 $P_{A \cup B}(X)$, and therefore, we have shown that the set $A \cup B$ is recursive.

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

This is false. We can disprove it by providing a counterexample: As we have shown in class, the set of all natural numbers is r.e. We also showed that the set $\neg K$ is not r.e.

Now let $A = \neg K$ and $B = N$. Then $A \subseteq B$ and $B$ is r.e., but $A$ is not r.e. End of proof.

(d) If $A \cup B$ is recursive, then both $A$ and $B$ are recursive.

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

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

If $A$ is recursive, we have a computable predicate $P_A(x)$ such that:

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

Now consider the following program:

$$Y \leftarrow \neg P_A(X)$$

Whenever $P_A(X)$ is 1, this program will return 0, and whenever $P_A(X)$ is 0, it will return 1. Therefore, this program computes the predicate $P_{\neg A}(x)$ that decides about membership in $\neg A$, and thus, $\neg A$ is recursive.

**Question 3: 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) $A = \{ x \in \mathbb{N} \mid x \text{ mod } 7 = 2 \}$

$A$ is recursive, because the following program computes the predicate $P_A(x)$ that tests for membership in $A$:
Z ← MODULO(X, 7)
IF Z ≠ 2 GOTO E
Y ← 1

(b) \( B = \{ x \in \mathbb{N} \mid x \text{ is the number of a program that computes the function } f(n) = 2n \} \)

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) \( C = \{ x \in \mathbb{N} \mid x \text{ is the number of a program that terminates on input 12 after at most 50 steps} \} \)

Thanks to the step-counter function we are able to write a program that computes a predicate \( P_C(x) \) for testing membership of \( x \) in \( C \):

\[
Y \leftarrow \text{STP}^{(1)}(12, X, 50)
\]

This program always terminates. Therefore, \( C \) is a recursive set.

(d) (Bonus Question) \( D = \{ x \in \mathbb{N} \mid x \text{ is the number of a program whose output is defined for at least one input} \} \)

\( D \) 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 let the main program terminate as well, and otherwise, 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. Only if we find such a termination, the main program will halt. Therefore, the main program will compute \( f_D(x) \). Here it is:

\[
\begin{align*}
[A] \quad & Z \leftarrow Z + 1 \\
& Z_2 \leftarrow \text{STP}^{(1)}(r(Z), X, l(Z)) \\
& \text{IF } Z_2 = 0 \text{ GOTO A}
\end{align*}
\]

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