Practice Exam Solutions

Question 1: ____ out of ____ points
Question 2: ____ out of ____ points
Question 3: ____ out of ____ points
Question 4: ____ out of ____ points
Question 5: ____ out of ____ points
Question 6: ____ out of ____ points
Question 7: ____ out of ____ points
Question 8: ____ out of _0_ points (bonus)

Total Score:

Grade:
Question 1: True or False?

Are the following statements true or false? Check the appropriate box for each statement. Notice that you will lose points for incorrect answers; you can leave both boxes blank if you are not sure which answer is correct.

true   false

a) A function $f$ is partially computable if and only if there is a program in the language $L_3$ that computes $f$. [X]   [ ]
b) For all sets $A$ and $B$, if $A \leq_m B$ and $B$ is r.e., then $A$ is also r.e. [X]   [ ]
c) If $A$ is r.e. and $\overline{A}$ is also r.e., then $A$ is always recursive. [X]   [ ]
d) $\text{UPCHANGE}_{2,10}(7) = 111$ [X]   [ ]
e) $\text{DOWNCHANGE}_{2,10}(1234) = 5$ [ ]   [X]
f) After a strict computation by a Post-Turing program, the tapehead is always on a blank symbol. [X]   [ ]
g) Any program in language $L_1$ can be translated into a Post-Turing program that computes the same function. [X]   [ ]
h) For all natural numbers $x$, $y$, and $z$ it is true that $\langle \langle x, y \rangle, z \rangle = \langle x, \langle y, z \rangle \rangle$. [ ]   [X]
i) For all languages $A$ and $B$, if $A \cap B$ is r.e., then $A$ and $B$ are both r.e. [ ]   [X]
j) Every total function is computable. [ ]   [X]
**Question 2: The Limping Turning Machine**

Imagine a Turing machine M' that works like a normal Turing machine except that whenever it moves to the left, it moves by two squares instead of one. Its movements to the right are by one square as usual. Give a detailed proof of the fact that M' can compute all functions that normal Turing machines can compute.

**Hint:** Think of the simulation of Turing machines with quintuple Turing machines that we discussed in class.

The basic idea is to show that whatever Turing machines can do, the machines of type M' can do as well. This would be achieved by devising a scheme for translating any given Turing machine into an equivalent machine of the new type. In other words, we want to show that we can simulate any given Turing machine with a machine of type M'.

To do this for a given Turing machine M with states $q_1, \ldots, q_k$, we take all quadruples in M of the types $q_i s_j s_m q_l$ and $q_i s_j R q_l$ and put them into our new machine M' without changing them at all. For each quadruple in M of the form $q_i s_j L q_l$, we add the quadruple $q_i s_j L^2 q_{k+1}$ to M', where $L^2$ stands for the double-leap to the left. Then we add new states $q_{k+1}, \ldots, q_{2k}$ to M', and also the quadruples $q_{k+i} s_j R q_j$ for all $i = 1, \ldots, k$, $j = 1, \ldots, n$, where $n$ is the number of symbols in the alphabet.

Then the machine M' will perform a computation equivalent to the one performed by M, and since we can do this for any Turing machine M, we have proven that the new type of machine is at least as computationally powerful as Turing machines.
Question 3: Programmer’s Proof

Show that if two sets $A$ and $B$ are both recursive, then $A - B$ is also recursive. In your proof, use a program in the language $L$ (possibly containing macros).

Since $A$ and $B$ are recursive, there are computable predicates $P_A$ and $P_B$ such that:

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

For our proof we have to show that there is a computable predicate $P_C$ so that

$$A - B = \{ x \in N \mid P_{A,B}(x) \}$$

We show this by writing a program that computes $P_{A,B}(x)$:

```
IF ~P_A(X) GOTO E
IF P_B(X) GOTO E
Y ← Y + 1
```
**Question 4: A Useful Program**

Describe in detail what the following program does. In which case would such a program be especially useful? Why?

[A] \[\text{IF } \text{STP}^{(1)}(X, p, T) \text{ GOTO C}\]

\[\text{IF STP}^{(1)}(X, q, T) \text{ GOTO E}\]

\[T \leftarrow T + 1\]

\[\text{GOTO A}\]

[C] \[Y \leftarrow 1\]

For a given input X, this program first simulates zero steps of the computation by the program with number p on input X and tests whether it terminates (only the empty program would terminate after zero steps). If so, the program above terminates with output one. If not, the program does the same simulation for program number q. If it terminates, the program above terminates with output zero, otherwise, the same as above is repeated, but this time one step is simulated. The number of simulated steps increases until either p or q terminates. At any point, if p terminates, the output is one, and if q terminates, the output is zero.

This program is useful if we know that either p or q will terminate on input X and want to determine which one it is. We cannot simulate the entire program execution of p and then the execution of q, because p may never terminate, and then our program would run forever without providing any conclusive result. We used this technique, called “dovetailing,” to show that whenever B and ¬B are r.e., then B is recursive.
Question 5: Enumeration

Explain in detail in your own words why the following theorem is true:

“A set $B$ is r.e. if and only if there is an $n$ for which $B = W_n$.”

$W_n$ is the domain of the partially computable function computed by the program with number $n$:

$$W_n = \{ x \in N \mid \Phi(x, n) \downarrow \}$$

So the infinite list $W_0, W_1, W_2 \ldots$ includes the domains of all partially computable functions.

Definition of r.e.: A set $B$ is r.e. just when it is the domain of a partially computable function $g(x)$:

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

So $B$ is r.e. if and only if $B$ is in the list $W_0, W_1, W_2 \ldots$, that is, if there is an $n$ for which $B = W_n$. 
Question 6: Recursive Enumeration

Show that the following set B is r.e.:

\[ B = \{ x \in \mathbb{N} \mid \text{program number } x \text{ halts on at least one input} \} \]

**Hint:** This question is a bit tricky, but it is a good exercise. Remember that the inverse pairing functions \( l(z) \) and \( r(z) \) go through every possible combination of integers as we increase \( z \) step by step from 0 to infinity.

In order to test whether \( x \) halts on any input, we cannot just let it run on input 0, 1, 2, and so on, because if, for example, it does not halt on 0, it will never return and so we will never get to test input 1.

So the idea is to use the step-counter program to run \( x \) on larger and larger input for more and more steps. If there is an input for which it halts after a finite number of steps, then eventually we will find it this way.

The trick for increasing both the input and the number of steps is to use the inverse pairing functions \( l(z) \) and \( r(z) \), shown as LEFT and RIGHT in the code (just like in our Haskell functions). When we start with \( z = 0 \) and increase \( z \) again and again, then \( l(z) \) and \( r(z) \) will eventually assume all possible combinations of integers. So we can use \( l(z) \) as the input to program number \( x \) and \( r(z) \) as the number of steps we want to simulate. Eventually we will test any input on any number of steps. Here is the program:

\[
\begin{align*}
\text{[A]} & \quad Z_2 \leftarrow \text{LEFT}(Z_1) \\
& \quad Z_3 \leftarrow \text{RIGHT}(Z_1) \\
& \quad \text{IF STP}(Z_2, X, Z_3) \text{ GOTO E} \\
& \quad Z_1 \leftarrow Z_1 + 1 \\
& \quad \text{GOTO A}
\end{align*}
\]

This program terminates if and only if program number \( X \) (its input) terminates on at least one input. Thus, this program can serve as the characteristic function for set \( B \), showing that \( B \) is r.e.
Question 7: String Theory

In our everyday decimal system, we use base \( n = 10 \) and the digits 0 to \((n - 1)\) to describe integers. However, when we developed the languages \( \mathcal{L}_n \) that process strings, we decided to use values 1 to \( n \) for individual symbols instead. Why did we not simply stick with the values 0 to \((n - 1)\)?

The problem with using symbols \( s_0, \ldots, s_{n-1} \) is that any integer is associated with more than one string representation. For example, using symbols \( s_0, \ldots, s_9 \), the integer 53 is associated with strings \( s_5s_3, s_0s_5s_3, s_0s_0s_5s_3, \) and so on. Our use of strings in programs relies on a one-to-one correspondence between strings and integers, and therefore this system is not suitable. Using symbols \( s_1, \ldots, s_{10} \), on the other hand, would associate integer 53 only with string \( s_5s_3 \), because there are no zero-value symbols that can be added to a string without changing the value of its associated integer.
Question 8 (Bonus): Yet Another Proof

Prove or disprove: Every finite set is recursive.

Let $B = \{b_1, \ldots, b_n\}$ be a finite set of cardinality $n$.

Then we can write the following program:

```
IF X = b_1 GOTO A
IF X = b_2 GOTO A
...
IF X = b_n GOTO A
GOTO E
[A]  Y ← Y + 1
```

This program always terminates and returns 1 if its input $X$ is in $B$ and returns 0 otherwise. We can use this program to provide a characteristic function for deciding about membership in $B$, showing that $B$ is recursive.