COMP218 **DECISION, COMPUTATION AND LANGUAGE**	**Tutorial 10:** Decidable languages.

1. **What is a reduction? Briefly explain how this technique can be used to prove that certain problems are undecidable.**

   Reduction is a technique where we reduce a problem to another. A reduction from L1 to L2 is a TM-computable function f such that

$$\forall w \in L_1 \; f(w) \in L_2$$
$$\forall w \notin L_1 \; f(w) \notin L_2$$

   The only way to prove a new problem L2 to be undecidable is to reduce a known undecidable problem L1 to L2.

2. **We can represent questions about context-free languages and regular languages by choosing a standard encoding for context-free grammars (CFG's) and another for regular expressions (RE's), and phrasing the question as recognition of the codes for grammars and/or regular expressions such that their languages have certain properties. Some sets of codes are decidable, while others are not.**

   In what follows, you may assume that G and H are context-free grammars with terminal alphabet {0,1}, and R is a regular expression using symbols 0 and 1 only. You may assume that the problem "Is L(G) = (0+1)*?", that is, the problem of recognizing all and only the codes for CFG's G whose language is all strings of 0's and 1's, is undecidable.

   There are certain other problems about CFG's and RE's that are decidable, using well-known algorithms. For example, we can test if L(G) is empty by finding the pumping-lemma constant n for G, and checking whether or not there is a string of length n or less in L(G). It is not possible that the shortest string in L(G) is longer than n, because the pumping lemma lets us remove at least one symbol from a string that long and find a shorter string in L(G).

You should try to determine which of the following problems are decidable, and which are undecidable:

- Is Comp(L(G)) equal to (0+1)*? [Comp(L) is the complement of language L with respect to the alphabet {0,1}.]
- Is Comp(L(G)) empty?
- Is L(G) intersect L(H) equal to (0+1)*?
- Is L(G) union L(H) equal to (0+1)*?
- Is L(G) finite?
- Is L(G) contained in L(H)?
- Is L(G) = L(H)?
- Is L(G) = L(R)?
- Is L(G) contained in L(R)?
- Is L(R) contained in L(G)?

Then, identify the true statement from the list below:

Answer	**Explanation**
○ "Is L(G) infinite?" is undecidable.	
○ "Is L(G) = L(H)?" is decidable.	
◉ "Is L(G) intersect L(H) equal to (0+1)*?" is undecidable.	Correct! The trick is to reduce "L(G) = (0+1)*" to this problem by choosing H to be a grammar whose language is (0+1)*.

○ "Is L(R) contained in L(G)?" is decidable.

Question Explanation

Here is the explanation for each of the problems:

- Is Comp(L(G)) equal to (0+1)*? This problem is the same as asking if L(G) is empty. It is therefore decidable.
- Is Comp(L(G)) empty? This problem asks if L(G) = (0+1)*; it is undecidable according to the facts given in the problem statement.
- Is L(G) intersect L(H) equal to (0+1)*? Undecidable. Proof: if we could decide this problem, let H be a CFG that generates (0+1)*, specifically S->0S|1S|ε. If L(G) intersect L(H) = (0+1)*, then L(G) = (0+1)*. That is, we could decide if L(G) = (0+1)*. But we are given that we can't decide that question.
- Is L(G) union L(H) equal to (0+1)*? Let H be a CFG that generates an empty language, specifically, S->S. Then L(G) union L(H) = (0+1)* if and only if L(G) = (0+1)*, and the argument proceeds as for the previous problem.
- Is L(G) finite? Let n be the pumping-lemma constant for G. If there is a string of length between n and 2n, then we can pump this string to generate an infinite number of strings. But if there is an infinite number of strings, then there must be one whose length is between n and 2n. If not, let z be the shortest string of length greater than 2n; there must be one if there are an infinite number of strings. The pumping lemma must let us remove between 1 and n characters of z and get another string in L(G). But this string must be between n and 2n in length. It can't be shorter than n, because z is longer than 2n. It can't be longer than 2n, because z is the shortest such string. Note that this test tells us both whether L(G) is finite and whether it is infinite.
- Is L(G) contained in L(H)? Suppose we could decide this question. Choose G so that L(G) = (0+1)*. Then the condition is true if and only if L(H) = (0+1)*, which we know is undecidable.
- Is L(G) = L(H)? The same argument applies if we choose H such that L(H) = (0+1)*.
- Is L(G) = L(R)? Again, the same argument applies if we let R be (0+1)*.
- Is L(G) contained in L(R)? Construct from G a CFG G', whose language is L(G) intersect Comp(L(R)). We can make this construction because there is an algorithm to find a representation (RE, DFA, or NFA) of the complement of a regular language, given a representation for the language itself, and also because there is a construction to find a CFG for the intersection of the languages of a CFG and RE (convert to a PDA and a DFA, respectively, and simulate the two in parallel; then convert the resulting PDA to a CFG). Now, L(G) is contained in L(R) if and only if L(G') is empty. We can test emptiness of a CFG, as described in the statement of the question.
- Is L(R) contained in L(G)? Choose R = (0+1)*. Then L(R) is contained in L(G) if and only if L(G) = (0+1)*, which we know is undecidable.

3. **For the purpose of this question, we assume that all languages are over input alphabet {0,1}. Also, we assume that a Turing machine can have any fixed number of tapes.**

   Sometimes restricting what a Turing machine can do does not affect the class of languages that can be recognized --- the restricted Turing machines can still be designed to accept any recursively enumerable language. Other restrictions limit what languages the Turing machine can accept. For example, it might limit the languages to some subset of the recursive languages, which we know is smaller than the recursively enumerable languages. Here are some of the possible restrictions:

   1. Limit the number of states the TM may have.
   2. Limit the number of tape symbols the TM may have.
   3. Limit the number of times any tape cell may change.
   4. Limit the amount of tape the TM may use.
   5. Limit the number of moves the TM may make.
   6. Limit the way the tape heads may move.

   Consider the effect of limitations of these types, perhaps in pairs. Then, from the list below, identify the combination of restrictions that allows the restricted form of Turing machine to accept all recursively enumerable languages.

   Answer **Explanation**

○ Allow the TM to use only n^2tape cells when the input is of length n.	
○ Allow the TM to run for onlyn^2 moves when the input is of length n.	
[✓ Allow a tape cell to change its symbol only once.]	The trick is to simulate any given TM by writing its ID's in succession on a tape, perhaps using another tape to copy ID's. Then, you only need to change blanks to some other symbol, and then leave them unchanged forever.
○ Allow the TM to run for only 2^n moves when the input is of length n.	
Total	

Question Explanation

The key observation is that given any TM M, we can design a very restricted form of TM that simulates M by writing successive ID's on a tape. The simulating TM can run back and fourth on the tape, computing the symbols of the next ID, one at a time. Remember that, unless the symbol being copied is adjacent to the head, the symbol cannot change.
On the other hand, if we limit the amount of tape that the TM may use to any computable function of the input length, then we can accept only recursive languages. The reason is that after a while, the TM must repeat an ID, and if it hasn't accepted by then, we can conclude it never will. Likewise, if we limit the number of moves to any computable function, we can accept only recursive languages.

Finally, if we limit the tape heads to move in only one direction, then nothing it writes can ever affect what it does. Thus, the TM can be simulated by a finite automaton, no matter how many tapes the TM has.

4. **Explain what is meant by a recursive language, and what is meant by a recursively enumerable language. Which one of the two sets stands for decidable problems?**

The languages accepted by Turing machines are called recursively enumerable (RE), and the subset of RE languages that are accepted by a TM that always halts are called recursive. The set of recursive languages stands for the set of decidable problems.