Theory of Computation
Problem Set 9
Universidad Politecnica de San Luis Potosi

Please start solving these problems immediately, don't procrastinate, and work in study groups.

Please do not simply copy answers that you do not fully understand;

Advice: Please try to solve the easier problems first (where the meta-problem here is to figure out which are the easier ones). Don't spend too long on any single problem without also attempting (in parallel) to solve other problems as well. This way, solutions to the easier problems (at least easier for you) will reveal themselves much sooner (think about this as a "hedging strategy" or "dovetailing strategy").

Problem One: Programming Turing Machines (24 Points)

In lecture, we saw how to scale the **WB** language up to **WB6**, which supported multiple tracks, multiple stacks, multiple tapes, and multiple finite variables.* However, the **WB6** language still does all its works on tapes and tracks. This question asks you to add in new features to **WB6** to make it look more like a normal programming language.

Let's define the language **WB7** to be **WB6** with the introduction of a finite number of *counters*, variables that can hold arbitrary natural numbers. Initially, each counter holds the value 0. Specifically, **WB7** is **WB6** with the addition of the following commands:

- `C++`, which increments counter `C`;
- `C--`, which decrements the counter `C` (if `C` is already zero, `C` is unchanged); and
- `If Zero(C), go to L`; which goes to line `L` if counter `C` holds the value 0.

For example, the following is a **WB7** program that checks for balanced parentheses by tracking a counter of the number of open parentheses:

```
// Checks whether the input string is a string of balanced parentheses.  We
// maintain a counter C holding the number of unmatched open parentheses.  If
// C drops below zero or is nonzero at the end of the input, we reject.  Otherwise
// we accept.

// Start:
0: If reading B, go to Done.
1: If reading ), go to MatchClose.
2: C++.
3: Move right.
4: Go to 0.

// MatchClose:
5: If Zero(C), go to Rej.
6: C--.
7: Move right.
8: Go to 0.

// Done:
9: If Zero(C), go to Acc.
10: Reject.

// Rej:
11: Reject.

// Acc:
12: Accept.
```

Although the above program only uses a single counter `C`, it is possible for a **WB7** program to have multiple different counters (call them C_1, C_2, …, C_n).

i. Describe how to convert an arbitrary **WB7** program into a **WB6** program. Your description should address the following questions:

   1. In converting from **WB7** to **WB6**, will you need to introduce extra stacks, tracks, variables, tape symbols, or tapes? If so, how many of each will you need and why?
   2. In converting from **WB7** to **WB6**, will you need to set up the tapes, stacks, tracks, or variables in any way before beginning the program? If so, how and why?
   3. How will you translate the commands `C++`, `C--`, and `If Zero(C) Go to L` into equivalent **WB6** commands? You should give the specific command sequence with which you will replace these commands.

   Note that counters can hold arbitrarily large natural numbers, so you cannot implement them by using **WB3**-style variables (since the number of tape symbols is finite).

---

* Handout 18 lists the major features of each version of the **WB*n*** languages.

The counters from **WB7** only support increment, decrement, and zero-testing. Let's define **WB8** to be **WB7** with a few more operations defined on these counters:

- **C_1 := C_2**, which sets the value of counter C_1 to the value of C_2;
- **C_1 := C_2 + C_3**, which sets the value of counter C_1 to the value of $C_2 + C_3$;
- **If C_1 = C_2, go to L**, which jumps to line L if counters C_1 and C_2 have the same value.

In all of these operations, it is assumed that all the counters in an expression are different. Thus the expression `C := C` is illegal, as is `C := C + D`.

For example, here is a program that checks if the number of `2`s in a string is equal to the number of `0`s in a string times the number of `1`s in a string:

```
// Checks whether the number of 2s in the input string is the product of the number
// of 1s in the input string and the number of 0s in the input string.  It
// maintains five counters:
//   A: Number of 0s in the input
//   B: Number of 1s in the input
//   C: Number of 2s in the input.
//   D: Scratch space.
//   E: Computed value of A × B.
// If we find C = E, then the string is accepted.  Otherwise we reject.

// Start:
0: If reading B, go to Mul.
1: If reading 0, go to R0.
2: If reading 1, go to R1.
3: If reading 2, go to R2.

// R0:
4: A++.
5: Go to Start.

// R1:
6: B++.
7: Go to Start.

// R2:
8: C++.
9: Go to Start.

// Mul:
10: If Zero(A), go to Chk.
11: D := E.
12: E := D + B.
13: A--.
14: Go to Mul.

// Chk:
15: If E = C, go to Acc.
16: Reject.

// Acc:
17: Accept.
```

ii. Describe how to convert an arbitrary **WB8** program into a **WB7** program. Your description should address the following questions:

1. In converting from **WB8** to **WB7**, will you need extra stacks, tracks, variables, tape symbols, tapes, or counters? If so, how many of each will you need and why?
2. In converting from **WB8** to **WB7**, will you need to set up the tapes, stacks, tracks, variables, or counters in any way before beginning the program? If so, how and why?
3. How will you translate **C_1 := C_2**, **C_1 := C_2 + C_3**, and **If C_1 = C_2, go to L** into equivalent **WB7** commands? You should give the specific command sequence with which you will replace these commands.

Since every **WB6** program is automatically a **WB7** program and any **WB7** program is automatically a **WB8** program, your results from parts (i) and (ii) shows that a language is **RE** iff there is a **WB8** program for it. This means that Turing machines are equivalent to programs with multiple tapes, stacks, tracks, finite variables, and unbounded counters. Hopefully this gives you a better understanding of the Church-Turing thesis!

Problem Two: Nondeterministic Algorithms (20 points)

Nondeterministic Turing machines make it possible to solve many problems that are extremely difficult to solve deterministically.

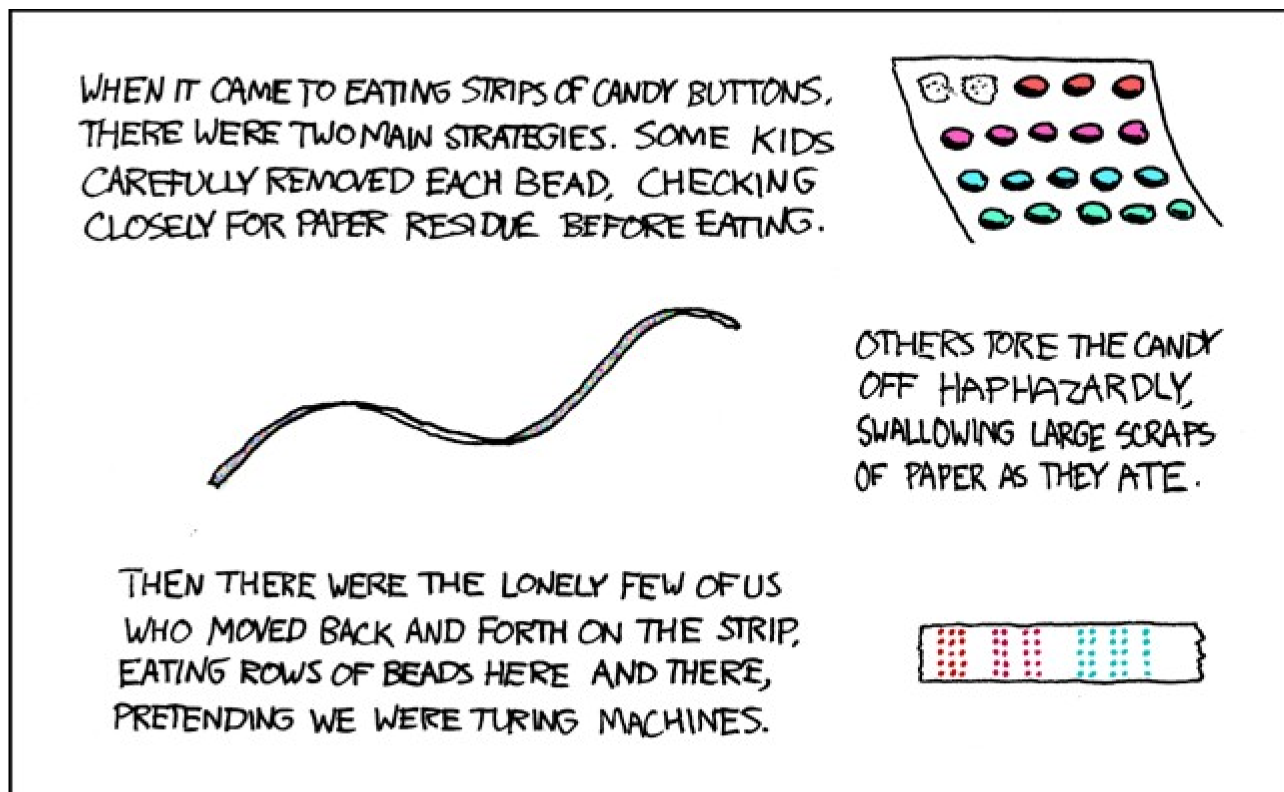
Prove each of the following by designing an appropriate nondeterministic Turing machine and proving that it has the appropriate language. Feel free to describe your NTMs using a high-level description similar to what we covered in lecture. Remember that for some NTM M, to prove that $\mathscr{L}(M) = L$, you should prove the following:

For any string $w \in \Sigma^*$, $w \in L$ iff there is some series of choices M can make such that M accepts w.

Notice that this statement is a biconditional.

i. Prove that the **RE** languages are closed under union. That is, if $L_1 \in$ **RE** and $L_2 \in$ **RE**, then $L_1 \cup L_2 \in$ **RE**.

ii. Prove that the **RE** languages are closed under concatenation. That is, if $L_1 \in$ **RE** and $L_2 \in$ **RE**, then $L_1L_2 \in$ **RE** as well.



http://xkcd.com/205/

We will cover the material necessary to solve these remaining problems in Monday's lecture.

Problem Three: Unrecognizable and Undecidable Languages (20 Points)

i. Prove or disprove: If L_1 is unrecognizable and $L_1 \subseteq L_2$, then L_2 is unrecognizable.

ii. Prove or disprove: If L_2 is unrecognizable and $L_1 \subseteq L_2$, then L_1 is unrecognizable.

iii. Do your answers still hold when "unrecognizable" is replaced with "undecidable?" What about "not context-free?" What about "not regular?"

Problem Four: A_{TM} is Unrecognizable? (12 Points)

In Monday's lecture, we proved that $A_{TM} \in \mathbf{RE}$ but that $A_{TM} \notin \mathbf{R}$. Our proof worked as follows:

- Assume, for the sake of contradiction, that A_{TM} is decidable.
- Using a decider for A_{TM} as a subroutine, construct a recognizer for L_D.
- Arrive at a contradiction, since we know that L_D is unrecognizable.
- Conclude, therefore, that A_{TM} is undecidable.

Initially, it might seem like we could easily modify this proof to show that $A_{TM} \notin \mathbf{RE}$ by simply changing our assumptions as follows:

- Assume, for the sake of contradiction, that A_{TM} is recognizable.
- Using a recognizer for A_{TM} as a subroutine, construct a recognizer for L_D.
- Arrive at a contradiction, since we know that L_D is unrecognizable.
- Conclude, therefore, that A_{TM} is unrecognizable.

Below is an incorrect proof along these lines. This proof contains an error that renders the proof invalid. Give the **<u>exact line</u>** of the proof that contains the error and explain why it is incorrect.

Theorem: A_{TM} is unrecognizable.

Proof: By contradiction; assume that A_{TM} is recognizable and let H be a recognizer for it. Then consider this machine D:

D = "On input $\langle M \rangle$:
  Construct $\langle M, \langle M \rangle\rangle$.
  Run H on $\langle M, \langle M \rangle\rangle$.
  If H accepts $\langle M, \langle M \rangle\rangle$, reject.
  If H rejects $\langle M, \langle M \rangle\rangle$, accept."

We claim that $\mathscr{L}(D) = L_D$. To see this, note that D accepts $\langle M \rangle$ iff H rejects $\langle M, \langle M \rangle\rangle$. Since H is a recognizer for A_{TM}, H rejects $\langle M, \langle M \rangle\rangle$ iff $\langle M, \langle M \rangle\rangle \notin A_{TM}$. Note that $\langle M, \langle M \rangle\rangle \notin A_{TM}$ iff $\langle M \rangle \notin \mathscr{L}(M)$, since $\langle M, \langle M \rangle\rangle$ is an encoding of a TM/string pair. Consequently, we have that D accepts $\langle M \rangle$ iff $\langle M \rangle \notin \mathscr{L}(M)$. Therefore, $\mathscr{L}(D) = L_D$.

Since $\mathscr{L}(D) = L_D$, we know that $L_D \in \mathbf{RE}$. But this is impossible, since we know that $L_D \notin \mathbf{RE}$. We have reached a contradiction, so our assumption must have been wrong. Thus A_{TM} is unrecognizable. ■

Problem Five: Why Decidability and Recognizability? (24 Points)

There are two classes of languages associated with Turing machines – the **RE** languages, which can be recognized by a Turing machine, and the **R** languages, which can be decided by a Turing machine.

Why didn't we talk about a model of computation that accepted just the **R** languages and nothing else? After all, having such a model of computation would be useful – if we could reason about automata that just accept recursive languages, it would be much easier to see what problems are and are not decidable.

It turns out, interestingly, that there is no class of automata with this property, and in fact the only way to build automata that can decide all recursive languages is to also have those automata also accept some languages that are **RE** but not **R**. This problem explores why.

Suppose, for the sake of contradiction, that there is a type of automaton called a **deciding machine** (or DM for short) that has the computational power to decide precisely the **R** languages. That is, $L \in \mathbf{R}$ iff there is a DM that decides L.

We will make the following (reasonable) assumptions about deciding machines:

- Any recursive language is accepted by some DM, and each DM accepts a recursive language.
- Since DMs accept precisely the recursive languages, all DMs halt on all inputs. That is, all DMs are deciders.
- Since deciding machines are a type of automaton, each DM is finite and can be encoded as a string. For any DM D, we will let the encoding of D be represented by $\langle D \rangle$.
- DMs are an effective model of computation. Thus the Church-Turing thesis says that the Turing machine is at least as powerful as a DM. Thus there is some Turing machine U_D that takes as input a description of a DM D and some string w, then accepts if D accepts w and rejects if D rejects w. Note that U_D can never loop infinitely, because D is a deciding machine and always eventually accepts or rejects. More specifically, U_D is the decider "On input $\langle D, w \rangle$, simulate the execution of D on w. If D accepts w, accept. If D rejects w, reject."

Unfortunately, these four properties are impossible to satisfy simultaneously.

i. Consider the language $REJECT_{DM} = \{ \langle D \rangle \mid D \text{ is a DM that rejects } \langle D \rangle \}$. Prove that $REJECT_{DM}$ is decidable.

ii. Prove that there is no DM that decides $REJECT_{DM}$.

Your result from (ii) allows us to prove that there is no class of automaton like the DM that decides precisely the **R** languages. If one were to exist, then it should be able to decide all of the **R** languages, including $REJECT_{DM}$. However, there is no DM that accepts the decidable language $REJECT_{DM}$. This means that one of our assumptions must have been violated, so at least one of the following must be true:

- DMs do not accept precisely the **R** languages, or
- DMs are not deciders, or
- DMs cannot be encoded as strings (meaning they lack finite descriptions), or
- DMs cannot be simulated by a TM (they are not effective models of computation)

Thus there is no effective model of computation that decides just the recursive languages.

Problem Six: Why All This Matters (20 Points)

The undecidability of the halting problem has enormous practical implications. This question explores some of them.

Since their memory is finite, computers are not as powerful as Turing machines. However, as computers start to get more and more memory, we can think of them as getting progressively closer and closer in power to Turing machines. For the purposes of this question, we'll assume that a standard computer is equivalent in power to a Turing machine.

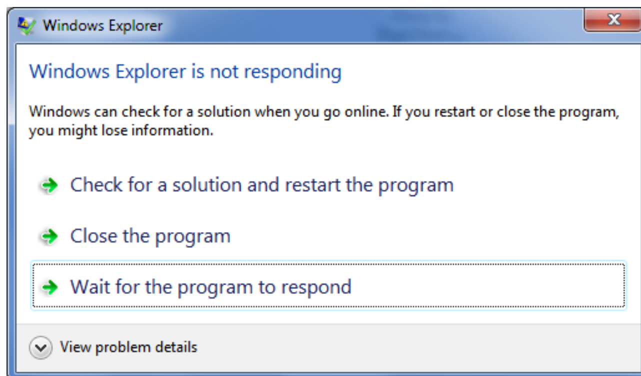
Because Turing machines and equivalently powerful models of computation can simulate one another, it is possible in any reasonable programming language to write a function like this one:

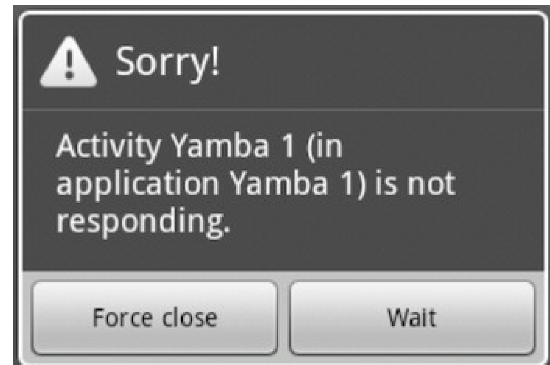
```
boolean simulateTuringMachine(TM M, string w)
```

This function takes in a suitably-encoded Turing machine M and a string w, then runs M on w. If M accepts w, then this function returns **`true`**. If M rejects w, then this function returns **`false`**. If M loops on w, then this function loops forever and never returns.

Using the existence of the above function, and the fact that a TM can simulate a computer, answer the following questions.

i. Most operating systems provide some functionality to detect programs that are looping infinitely. Typically, they display a dialog box containing a message like these shown below:





These messages give the user the option to terminate the program or to let the program keep running in the hopes that it stops looping.

An ideal OS would shut down any program that had gone into an infinite loop, since these programs just waste system resources (processor time, battery power, etc.) that could be better spent by other programs. Since it makes more sense for the OS to automatically detect programs that have gone into an infinite loop, why does the OS have to ask the user whether to terminate the program or let it keep running? Justify your answer.

ii. Suppose that you want to create a website that teaches people how to program. On this site, you give a set of programming problems and invite users to submit programs that solve those problems. For each programming problem, you write a small set of test cases that submitted programs should be able to pass. Each test cases consists of a set of inputs to the user's program, along with the expected outputs. Prove that it is impossible to automatically verify whether an arbitrary submitted program passes these tests. You can assume that all that matters is whether the program eventually outputs the right answer, not how long it takes to do so.

iii. Suppose that you want to build an optimizing compiler that takes as input a program and produces as output the smallest possible program equivalent to it. For example, given a program like this one:

```
int main() {
    int sum = 0;
    for (int i = 0; i < 10; i++) {
        sum += i;
    }
    cout << sum << endl;
}
```

The optimizer might output a program like this one:

```
int main() {
    cout << 45 << endl;
}
```

Prove that it is impossible to write a program that automatically optimizes any input program.

Problem Seven: Course Feedback (5 Points)

We want this course to be as good as it can be, and we'd really appreciate your feedback on how we're doing. For a free five points, please answer the following questions. We'll give you full credit no matter what you write (as long as you write something!), but we'd appreciate it if you're honest about how we're doing.

i. How hard did you find this problem set? How long did it take you to finish? Does that seem unreasonably difficult or time-consuming for a five-unit class?

ii. Did you attend Monday's problem session? If so, did you find it useful?

iii. How is the pace of this course so far? Too slow? Too fast? Just right?

iv. Is there anything in particular we could do better? Is there anything in particular that you think we're doing well?

Extra Credit Problem: Unrestricted Grammars (5 Points)

An *unrestricted grammar* is a substantial generalization of context-free grammars in which arbitrary strings can appear on the left-hand side of a production, not just nonterminals. This allows unrestricted grammars to replace arbitrary substrings of a sentential form with new substrings.

For example, the following unrestricted grammar generates the language $\{ \mathtt{0}^n\mathtt{1}^n\mathtt{2}^n \mid n \in \mathbb{N} \}$:

$$\begin{gathered} S \to \mathtt{0}SX \\ X \to A\mathtt{2} \\ \mathtt{2}A \to A\mathtt{2} \\ SA \to \mathtt{1} \\ \mathtt{1}A \to \mathtt{11} \end{gathered}$$

One possible derivation of 000111222 is shown here:

$$\begin{array}{ll}
 \underline{S} & \text{(Apply } S \to \mathtt{0}SX\text{)} \\
\Rightarrow \mathtt{0}\underline{S}X & \text{(Apply } S \to \mathtt{0}SX\text{)} \\
\Rightarrow \mathtt{00}\underline{S}XX & \text{(Apply } S \to \mathtt{0}SX\text{)} \\
\Rightarrow \mathtt{000}S\underline{X}XX & \text{(Apply } X \to A\mathtt{2}\text{)} \\
\Rightarrow \mathtt{000}SA\mathtt{2}\underline{X}X & \text{(Apply } X \to A\mathtt{2}\text{)} \\
\Rightarrow \mathtt{000}SA\mathtt{2}A\mathtt{2}\underline{X} & \text{(Apply } X \to A\mathtt{2}\text{)} \\
\Rightarrow \mathtt{000}SA\mathtt{2}A\underline{\mathtt{2}A}\mathtt{2} & \text{(Apply } \mathtt{2}A \to A\mathtt{2}\text{)} \\
\Rightarrow \mathtt{000}SA\underline{\mathtt{2}A}A\mathtt{22} & \text{(Apply } \mathtt{2}A \to A\mathtt{2}\text{)} \\
\Rightarrow \mathtt{000}SAA\underline{\mathtt{2}A}\mathtt{22} & \text{(Apply } \mathtt{2}A \to A\mathtt{2}\text{)} \\
\Rightarrow \mathtt{000}\underline{SA}AA\mathtt{222} & \text{(Apply } SA \to \mathtt{1}\text{)} \\
\Rightarrow \mathtt{000}\underline{\mathtt{1}A}A\mathtt{222} & \text{(Apply } \mathtt{1}A \to \mathtt{11}\text{)} \\
\Rightarrow \mathtt{0001}\underline{\mathtt{1}A}\mathtt{222} & \text{(Apply } \mathtt{1}A \to \mathtt{11}\text{)} \\
\Rightarrow \mathtt{000111222} &
\end{array}$$

Given an unrestricted grammar G, we define $\mathscr{L}(G) = \{ w \in \Sigma^* \mid S \Rightarrow^* w \}$.

Prove that a language L is **RE** iff there is an unrestricted grammar G where $\mathscr{L}(G) = L$. This gives a formalism for describing the **RE** languages through generation, just as the Turing machine is a formalism for describing the **RE** languages through recognition.