CSE 311: Foundations of Computing I

Homework 7 (due November 26th at 11:59 PM)

Directions: *Write up carefully argued solutions to the following problems. Your solution should be clear enough that it should explain to someone who does not already understand the answer why it works. You may use results from lecture, the theorems handout, and previous homeworks without proof.*

1. Express Yourself [Online] (25 points)

For each of the following, construct regular expressions that match the given set of strings:

(a) [5 Points] Binary strings with length divisible by 3.

(b) [5 Points] Binary strings where **every** occurrence of a 0 is immediately followed by a 1.

(c) [5 Points] Binary strings where **no** occurrence of a 11 is immediately followed by a 0.

(d) [5 Points] Binary strings that start with 1 and have even length.

(e) [5 Points] Binary strings with an odd number of 1s.

Submit and check your answers to this question here:

https://grinch.cs.washington.edu/cse311/regex

You **must also** submit a screenshot of your submitted regexes in Canvas.

2. Grammar School (15 points)

For each of the following, construct context-free grammars that generate the given set of strings. If your grammar has more than one variable, we will ask you to write a sentence describing what sets of strings you expect each variable in your grammar to generate.

For example, if your grammar were:

$$\begin{aligned} \mathbf{S} &\to \mathbf{E} \mid \mathbf{O} \\ \mathbf{E} &\to \mathbf{EE} \mid \mathbf{CC} \\ \mathbf{O} &\to \mathbf{EC} \\ \mathbf{C} &\to 0 \mid 1 \end{aligned}$$

We would expect you to say "E generates (non-empty) even length binary strings; O generates odd length binary strings; C generates binary strings of length one."

(a) [5 Points] Binary strings with an odd number of 0s.

(b) [5 Points] $\{0^{m+n}1^{2n}0^m : m, n \geq 0\}$

(c) [5 Points] All sum-of-products boolean expressions over the variables a, b, c that contain at least one minterm. Use $\overline{x}$ to denote the negation of the variable x, and make the product symbol $\cdot$ explicit in the expressions. See Lecture 05 for the properties of the sum-of-products form.

3. Friends and Relations (15 points)

Consider the set of all Twitter users. In each part of this problem, we define a relation R on this set. For each one, state whether R is or is not reflexive, symmetric, antisymmetric, and/or transitive. (No proofs.)

(a) [5 Points] $(a, b) \in R$ if b did not join Twitter before a.

(b) [5 Points] $(a, b) \in R$ if there are no common followers of a and b.

(c) [5 Points] $(a, b) \in R$ if every user who is a follower of a is also a follower of b.

4. Public Relations (10 points)

Let A be a set. Let R and S be transitive relations on A.

(a) [5 Points] Is $R \cup S$ necessarily transitive? If so, give an English proof. If not, give a counterexample.

(b) [5 Points] Is $R \cap S$ necessarily transitive? If so, give an English proof. If not, give a counterexample.

5. Substitute Teacher (35 points)

When studying modular arithmetic, we made the claim that, if we replace a variable in an arithmetic expression with a value that it is congruent to, the resulting expression will be congruent to the one we started with. We now have sufficient tools to prove that, and we will do so in this problem.

We start by formalizing arithmetic expressions. Consider the set **Expr**, representing parse trees for arithmetic expressions, defined recursively as follows:

Bases Step: Variable(x) is in **Expr**, representing the variable x, and for any $v \in \mathbb{Z}$, Number(v) is in **Expr**, representing the integer v.

Recursive Step: For any s and t in **Expr**, Plus(s, t) is in **Expr**, representing $s + t$, and Times(s, t) is in **Expr**, representing $s \cdot t$.

For example, the expression "3 + 4x" would become Plus(Number(3), Times(Number(4), Variable(x))). Every possible arithmetic expression (with a single variable x) can be represented by an element of **Expr**.

If we choose a specific, integer value $v \in \mathbb{Z}$ for the variable x, then we can calculate the value of the entire expression, with $x = v$, recursively, as follows:

$$\begin{array}{rcll} \text{value}_v(\text{Number}(w)) & = & w & \forall\, w \in \mathbb{Z} \\ \text{value}_v(\text{Variable}(x)) & = & v & \\ \text{value}_v(\text{Plus}(a,b)) & = & \text{value}_v(a) + \text{value}_v(b) & \forall\, a, b \in \textbf{Expr} \\ \text{value}_v(\text{Times}(a,b)) & = & \text{value}_v(a) \cdot \text{value}_v(b) & \forall\, a, b \in \textbf{Expr} \end{array}$$

Now, we define the function subst_c, which substitutes the expression $c \in$ **Expr** for all instances of the variable x in an arithmetic expression, resulting in a new **Expr**, as follows:

$$\begin{array}{rcll} \text{subst}_c(\text{Number}(w)) & = & \text{Number}(w) & \forall\, w \in \mathbb{Z} \\ \text{subst}_c(\text{Variable}(x)) & = & c & \\ \text{subst}_c(\text{Plus}(a,b)) & = & \text{Plus}(\text{subst}_c(a), \text{subst}_c(b)) & \forall\, a, b \in \textbf{Expr} \\ \text{subst}_c(\text{Times}(a,b)) & = & \text{Times}(\text{subst}_c(a), \text{subst}_c(b)) & \forall\, a, b \in \textbf{Expr} \end{array}$$

With those definitions in place, we want to prove:

$$\forall\, v \in \mathbb{Z}.\, (\text{value}_v(\text{Variable}(x)) \equiv \text{value}_v(c) \pmod m) \;\rightarrow\; \forall\, a \in \textbf{Expr}.\, \text{value}_v(a) \equiv \text{value}_v(\text{subst}_c(a)) \pmod m),$$

where $m > 0$ is an integer and $c \in$ **Expr** is a fixed arithmetic expression. In English, this says, for any value we might pick for x, if the value of the expression c is congruent to that of x, the value of any expression a that uses the variable x is congruent to the value of the same expression with c substituted for x. (I.e., if we replace every instance of x in the expression a by c, we get an expression whose value is congruent to that of a.)

For example, if c is the **Expr** for $x+3m$, then the claim says that we can replace x by $x+3m$ in any **Expr** and the result will be an Expr whose value is congruent to the original one, modulo m, since $x \equiv x + 3m \pmod m$. For example, we could replace x by $x+3m$ in the **Expr** for $5(x+7)$ to get the **Expr** for $5((x+3m)+7)$, which would simplify to $5x + 15m + 7$. This second expression is not the same as the first, but their difference, $15m$, disappears when we work modulo m.

Prove the claim above using structural induction over $a \in$ **Expr**.

Hint: As in problem 6.4, my solution is shorter than the problem statement! This proof also is all "easy parts". It requires no deep insights. To prove it, just follow the structure of the claim we're trying to prove....

6. Extra Credit: Ambiguity (0 points)

Consider the following context-free grammar.

⟨**Stmt**⟩ → ⟨**Assign**⟩ | ⟨**IfThen**⟩ | ⟨**IfThenElse**⟩ | ⟨**BeginEnd**⟩
⟨**IfThen**⟩ → if condition then ⟨**Stmt**⟩
⟨**IfThenElse**⟩ → if condition then ⟨**Stmt**⟩ else ⟨**Stmt**⟩
⟨**BeginEnd**⟩ → begin ⟨**StmtList**⟩ end
⟨**StmtList**⟩ → ⟨**StmtList**⟩⟨**Stmt**⟩ | ⟨**Stmt**⟩
⟨**Assign**⟩ → `a := 1`

This is a natural-looking grammar for part of a programming language, but unfortunately the grammar is "ambiguous" in the sense that it can be parsed in different ways (that have distinct meanings).

(a) [0 Points] Show an example of a string in the language that has two different parse trees that are meaningfully different (i.e., they represent programs that would behave differently when executed).

(b) [0 Points] Give **two different grammars** for this language that are both unambiguous but produce different parse trees from each other.