CS 3100, Spring 2020, Asg-7, 100 pts
Given 4/1/20 – Due 4/17/20 11:59pm

NAME: **UNID:**

1. **(25 pts)** Do Problem 5, Page 232, Exercise 15.2.3 which asks you to perform a mapping reduction from the PCP to the CFG Ambiguity problem, and thus argue that if there is a decider for the CFG Ambiguity problem, then there will be a decider for PCP also.

   (a) Here is how we can build a mapping reduction from PCP to CFG grammar ambiguity; please fill in missing steps (if any) and argue that the mapping reduction actually works (achieves its purpose).

   Let
   $$A = w_1, w_2, \ldots, w_n$$
   and
   $$B = x_1, x_2, \ldots, x_n$$
   be two lists of words over a finite alphabet Σ. Let $a_1, a_2, \ldots, a_n$ be symbols that do not appear in any of the w_i or x_i. Let G be a CFG
   $$(\{S, S_A, S_B\},\ \Sigma \cup \{a_1, \ldots, a_n\},\ P,\ S),$$
   where P contains the productions

   $S \to S_A$,
   $S \to S_B$,
   For $1 \le i \le n$, $S_A \to w_i S_A a_i$,
   For $1 \le i \le n$, $S_A \to w_i a_i$,
   For $1 \le i \le n$, $S_B \to x_i S_B a_i$, and
   For $1 \le i \le n$, $S_B \to x_i a_i$.

   Now, argue that G is ambiguous if and only if the PCP instance (A, B) has a solution (thus, we may view the process of going from (A, B) to G as a mapping reduction). □

State all the necessary assumptions and steps. Your answer must consist of these:

(a) (7 points) Definition 15.5 mathematically defines the term *mapping reduction.* Clearly point out the full extent of the meaning of the "if and only if" argument, taking the illustration in Figure 15.8 as an example. More specifically,

   i. (4 points) $x \in A \Rightarrow f(x) \in B$ is one part of Definition 15.5. Refer to Figure 15.8 and describe how this part of the definition plays out. In other words, assume that as shown in Figure 15.8, it is easy to describe a single Turing machine (call it `Translate`) that implements the $f()$ function. The job of `Translate` is to print out M' and w on the tape. Describe the "body" (algorithm) of `Translate` **in just three sentences** (because a more detailed question is coming later. *You are allowed to read, understand, and then incorporate the explanations given in Figure 15.8.* You can say "IF M' when run on w does [THIS], then M' when run on w does [THIS]," as the punchline in your 3-sentence answer. The `THIS` is the key thing in your answer.

ii. (3 points) $x \notin A \Rightarrow f(x) \notin B$ is another part of Definition 15.5. Again, as above, you can say "IF M' when run on w does not do [THIS], then M' when run on w does not do [THIS]," as the punchline in your 3-sentence answer. The `THIS` is the key thing in your answer.

(b) (3 points) Mathematically define the language *Amb*—the language of ambiguous CFGs (take ideas and notations from Asg-6, including the $\langle \ldots \rangle$ notation). Your answer must define an actual *language* that *Amb* is supposed to be.

(c) (15 points) Using the mapping reduction proposed in Exercise 15.2.3, argue that if the PCP system has a solution, then the mapped object—the CFG—is ambiguous. This is a *key* part of your answer. It asks you to argue as follows in your answer:

- (10 points for elaborating on PCP having a solution) "Suppose the given PCP system **has** a solution (**Hint:** the dominoes are (w_i, x_i)). Then a direct consequence of this is [THIS]." Here, in your answer, you have to say what it means for a *common* string being parsed using the grammar resulting from this mapping reduction. Elaborate on "[THIS]" in 3-4 tight sentences.
- (5 points for elaborating on parsing ambiguity) The [THIS] above must result in a conclusion that matches the definition of mapping reductions for this problem. That is, the translated grammar must have [THIS] property with respect to ambiguity. Perfectly well understand why the grammar was obtained from this PCP instance in the very specific manner shown.

(d) (5 points for arguing that no solution means THIS for ambiguity) Show that if the PCP system has no solution, then the mapped object (CFG) is not ambiguous. Again, a "mirror" of the above answer showing that **no sentence** in your grammar will have an ambiguous parse. Again refer to the nature of the grammar produced, and finish this part of the proof.

2. **(20 points)** (Mapping reduction for CFL_{TM} in C-style)

Show via a mapping reduction from A_{TM} that CFL_{TM} is not decidable, where

$$CFL_{TM} = \{\langle M \rangle \ : \ M \text{ is a TM whose language is context free}\}$$

FIRST STEP: Study the proof of $Regular_{TM}$ being undecidable (see below). It is also described in our book around Figure 15.10. It can also be shown in C-style as shown below.

YOUR TASK: Write a very similar proof for CFL_{TM} in

(a) (10 points) the mathematical style, and

(b) (10 points) the C style

Proof of undecidability of $Regular_{TM}$: Mathematical Style

Define

$$Regular_{TM} = \{\langle M \rangle \ : \ M \text{ is a } \mathrm{T}M \text{ whose language is regular}\}$$

```
/*  Now define machine M' */

    M'(x) {
      if x is of the form 0^n 1^n then goto accept_M' ;
      Run M on w ;
      If M accepts w, goto accept_M' ;
      If M rejects w, goto reject_M' ; }
```

$$Decider_{Regular_{TM}}(M') = \begin{cases} accepts & \Rightarrow L(M') \ is \ regular \\ \quad \Rightarrow \ Language \ is \Sigma^* & \Rightarrow M \ accepts \ w \\ rejects & \Rightarrow L(M') \ is \ not \ regular \\ \quad \Rightarrow \ Language \ is \ 0^n 1^n & \Rightarrow M \ does \ not \ accept \ w \end{cases}$$

Figure 1: Mapping reduction from A_{TM} to $Regular_{TM}$

We can prove $Regular_{TM}$ to be undecidable by building the Turing machine M' via mapping reduction , as shown in Figure 1. as shown in Figure 1. The proof argument is captured in $Decider_{Regular_{TM}}(M')$ in Figure 1.

Proof of undecidability of $Regular_{TM}$: C-Style

The same proof idea above can be shown in C code. Let us assume that we are working in C (or pseudo-C syntax). Let us introduce some types.

- `string s`: means `s` is of type string.
- `int i`: means `i` is of type int.
- `TM M`: means `M` is of type Turing Machine.
- C allows format strings such as `%s` and `%c`. I'll use `%s1` and `%s2` if I feed in multiple strings.
- `CProg C`: means `C` is of type `CProg` ("C program")
- We assume that anything of type `TM` is also a `C` program.
- Note that in C, when we say `int`, we mean "the set of integers."
- Thus, `int i` means `i` is a member of the set `int`.
- Likewise, `ATM <M, w>`. This means that `ATM` is a type, or set, and `<M,w>` are members of this set `ATM`.
- Thus when you see A_{TM}, imagine the type or set `ATM`. Its members are pairs `<M,w>`.
- You also know by now that the members of `ATM` are pairs of the form `<CProg, string>`. That is, `<M,w>` is nothing but something that has type `<CProg, string>`.

Let us introduce these C functions that are allegedly in some C library:

- `function SemiDeciderATM()`: This is a semi-decider for the set (or type) `ATM`. It is not a full decider but only a semidecider.
- `function FullDeciderATM()`: This is a full decider for the set (or type) `ATM`. **We have mathematically shown** that `function FullDeciderATM()` cannot exist.
- `function FullDeciderRegTM()`: This is a full decider for the set (or type) `REGTM`. This is the set of TMs whose languages are a regular set. **We are going to show that** if `function FullDeciderRegTM()` exists, then `function FullDeciderATM()` will end up existing. This will result in a contradiction.

Now define a translator from TMs to TMs (or CProg to CProg) called function `Translate`. This is our mapping reduction function! Function `Translate` must take `<M,w>` and produce another `CProg`.

```
CProg Translate(CProg M, string w) { // This is the mapping reduction function
 ... body of Translate ...
}
```

Our goal is to design `CProg Translate(...)` in such a way that it tries to trick `FullDeciderRegTM()` if it were to exist. For that, the design of `CProg Translate` must hide within it the power to define `FullDeciderATM()`.

We design things so that `Translate()` prints out `M'` by splicing-in `M` and `w`.

```
CProg Translate(CProg M, string w) { // This is the mapping reduction function
 printf
   ("
    M'(x) {
      if x is of the form 0^n 1^n then goto accept_M';

      Run %s1 on %s2 ; // First %s1 will splice-in M, second %s2 will splice-in w

      If this execution results in %s1 accepting %s2,
      then M' goes to accept_M';

      If %s1 rejects %s2, then M' goes to reject_M';

    }", M, w);

}
```

How does `Translate()` help?

Notice that `Translate()` is our mapping reduction thingie. It takes an `M` and `w` and splices it into a print statement, where the **entire** print statement's body is the `M'` function!

Now if the output of `Translate()` – which is the `M'` machine – is presented to `FullDeciderRegTM()` – if it were to exist – then we have essentially created `FullDeciderATM()`. This is a contradiction.

Here is how `FullDeciderATM()` will work then (can be written in Pseudo-C).

```
CProg FullDeciderATM(CProg M, string w) { // This is the mapping reduction function

 return FullDeciderRegTM( Translate(M,w) ) ;

}
```

WHY DOES THIS PROOF WORK?

The only way the language of the `M'` machine will be deemed regular is if `M` accepts `w`. Else its language can't be regular. Thus this "secret" is revealed by judging that `M'` has a regular language.

3. **(25 points)** (Running BDD tools to understand NP-completeness reductions)

   (a) (5 pts) Consider Figure 16.9 of the book. Describe why the mapping reduction from the formula ϕ shown in this figure to the clique proves that if ϕ is false, there is no 4-clique? **Answer in two clear sentences.** Any two sentences are fine so long as it reflects your understanding well.

   Answer: ...

   (b) (5 pts) Drop the last conjunct of ϕ, calling the formula ϕ_1. Enter the resulting ϕ_1 into the BDD tool that can be launched via `notebooks/driver/Drive_BDD_Illustration.ipynb`. Obtain the satisfying assignment for ϕ_1. What is the satisfying assignment? **Include the BDD diagram** by saving the BDD image and including it in your answer. Save by right-clicking on the image. **Answer in terms of the values of `x1` and `x2`.**

   Answer: ...

   (c) (5 pts) Now draw the corresponding clique for ϕ_1.

   Answer: ...

   (d) (5 pts) Now explain from the satisfying assignment for ϕ_1 and the clique for ϕ_1 that given that this formula is now satisfiable, there is a 3-clique (*i.e.*, a triangle). Describe the 3-clique as you would describe any graph: as a pair $(nodes, edges)$, where *nodes* is a **set** of nodes, and *edges* is a set of *pairs* of nodes. Also, provide a drawing of this 3-clique.

   Answer: ...

   (e) (5 pts) Suppose someone comes up with a P-time solver for cliques. How does this allow you to obtain a P-time solver for 3-SAT? Describe

   in **two clear sentences** reflecting your understanding. Use any two sentence forms to express: we just want to see how you are thinking.

   Answer: ...

4. **(20 points)** Boolean satisfiability is **one of the most important** of CS algorithms, for many reasons:

   - The first NP-complete problem was precisely this (SAT)
   - Solving SAT efficiently will result in solving a host of problems efficiently
   - Despite the hardness of SAT, it is a "workhorse" tool, finding uses in all kinds of program analysis tools, computer security tools, etc.

   Things have advanced so much that as undergrads taking a basic models of computation class, you can run a fairly advanced SAT tool in Javascript – *in your browser.* **How lucky can you get ?!**

   This problem asks you to get a taste of running a SAT tool and seeing how things are encoded. Specifically, you will be running CryptoMinisat on a SAT formula. You don't need to install this tool: merely go to page 265 of our book, consult Figure 16.10, and presto—there is a link to this tool that you can click! When you do this, the tool comes up with a prefilled formula. There is a Play button that you can click whereupon it solves the SAT instance.

   This assignment asks you to replace this SAT instance with something bigger: specifically, the Pigeonhole problem (`hole6.cnf`) from `https://people.sc.fsu.edu/~jburkardt/data/cnf/cnf.html`. Just click the above link, and get the `hole6.cnf` file, and plunk the CNF into the buffer.

   Hit "play" and report on the execution time (you can look at your phone's clock). If under 2 seconds, say "negligible" for your answer!

   How much time would such a problem take through brute-force enumeration of 2^n combinations on a computer that takes a microsecond per variable combination (the n is the number of variables used in the Pigeonhole problem)? **HINT:** Here is how you read the contents of a CNF file:

```
c File:  hole6.cnf <--- these are comment lines - starts wth a "C"
c...
c
p cnf 42 133     <--- CRUCIAL !! Tells you there are 42 variables and 133 clauses
-1     -7    0   <--- This line says (!x1 + !x7). The "0" is just end-of-a-clause marker!
-1     -13   0   <--- This line says (!x1 + !x13)
...
 12     11     10     9      8      7    0 <--- This clause reads
                                           (x12 + x11 + x10 + x9 + x8 + x7)
...
```

   ANSWER: OK now you have all the info you need to calculate the time it takes to enumerate 2^n combinations!!

   (a) (2 points) CryptoMinisat runtime;

   (b) (6 points) 2^n runtime estimation;

   (c) (12 points) List six facts that you found interesting in these articles:

https://cacm.acm.org/magazines/2009/8/34498-boolean-satisfiability-from-theoretical-hardness-to-practical-success/
fulltext

and

https://en.wikipedia.org/wiki/Boolean_satisfiability_problem

Anything that interested you is fine – theoretical or practical. Please offer 3 sentences per point that interested you.