CSE 555
Homework 2 Solutions

1a) The following TM \( R \) is a recognizer for \( DOUB_{CFG} \):
\[ R = \text{"On input } (G) \text{ where } G \text{ is a CFG}\]
- Let \( E \) be an enumerator for the regular (and thus recognizable) language \( \Sigma^* \) that prints each \( w \in \Sigma^* \)
in lexicographic order (length non-decreasing, alphabetic order).
- For each \( w \) that \( E \) prints, simulate \( G \) on \( ww \).
- If \( G \) accepts any \( ww \), accept.

1b) Idea: To arrive at a contradiction, with a decider for \( DOUB_{CFG} \), we will craft a decider for \( A_{TM} \). We want to design a PDA (which has an equivalent CFG) in the style of the proof in Sipser of \( ALL_{CFG} \) but instead of accepting every string when the TM \( M \) fails to accept \( w \), we will design it to accept \( w w \) (as well as some other strings) when \( M \) accepts \( w \). The approach of accepting the accepting computation history \( w \) twice will not work since it is not possible for a PDA to verify that \( C_1 \) yields \( C_2 \) and \( C_2 \) yields \( C_3 \) since the first verification consumes the input corresponding to \( C_2 \) so that it cannot be used for the comparison to \( C_3 \). The approach used instead is to check if \( C_{2i-1} \) yields \( C_{2i} \) while reading the first part of the string and check if \( C_{2i} \) yields \( C_{2i+1} \) while reading the second part of the string for integers \( i \). Now this will certainly accept strings of the form \( xy \) where every configuration checked happens to be correct but \( x \neq y \) but it will certainly accept \( w w \) where \( w \) is the accepting configuration for \( M \) on \( w \) when such a configuration exists.
Thus, deciding whether \( M \) accepts \( w \) is reduced to deciding whether the CFG equivalent to this PDA is a member of \( DOUB_{CFG} \).

Proof
We will understand computation histories in the same way as in the proof of \( ALL_{CFG} \) so that a computation history is \( \#C_1\#C_2\#C_3\#C_4\#\ldots\#C_l \) and to be able to tell the first part from the second part, we will use a delimiter \( \@ \) after each computation history. We design the PDA in a similar way to the proof of \( ALL_{CFG} \). First note that the if at any point the machine determines that the input is not in the form of a pair of computation histories, the machine immediately rejects. The PDA first checks the first configuration to verify that it is \( C_1 \) (pushing it onto the stack as well for the next step). It then pushes \( C_{2i-1} \) onto the stack, then compares with \( C_{2i} \) to check if they match except around the head position and there, the PDA checks if the difference follows the transition function of \( M \). Finally, if the last configuration is an accepting configuration \( C_l \), we move to the second string (ensuring that we see and consume the delimiter \( \@ \)). This time we do not need to check if the string starts with \( C_1 \) or ends with \( C_1 \) (since if the string is of the form \( w w \), the second \( w \) will also start with \( C_1 \) and end with \( C_1 \) but there is no harm in doing so, we push \( C_{2i} \) onto the stack and compare with \( C_{2i+1} \) to check if they match except around the head position and there, the PDA checks if the difference follows the transition function of \( M \).

Assume \( D \) is a decider for \( DOUB_{CFG} \). The following TM \( A \) is a decider for \( A_{TM} \):

\[ M = \text{"on input } (M,w) \text{ \"}
- Construct PDA \( P \) as detailed above.
- Convert the PDA to an equivalent CFG \( C \).
- Run \( D \) on \( (C) \).
- If \( D \) accepts, accept and if \( D \) rejects, reject.”

Thus we have a contradiction so \( DOUB_{CFG} \) is undecidable.

2) Idea: We can construct a grammar that operates similarly to the PDA in the Sipser proof of \( ALL_{CFG} \) that contains only one variable on the right side.

Proof
We craft a grammar as follows
\[ S \rightarrow \text{NonStart} | \text{NonAccept} | \text{NonTransition} | \epsilon \]
so that any string that is not a computation of $M$ on $w$ is generated; any string that does not start with $\#C_1\#$, does not end with $\#C_1\#$, or that contains an invalid transition (including intermediate strings that do not represent configurations).

The NonStart and NonAccept rules are relatively simple (though lengthy). We will use the following rules to denote any string over the alphabet $\Sigma$:

$$\text{ANY} \rightarrow z\text{ANY}$$

for each character we are using to encode the computation history $z$. Let $\#C_1\#$ denote the start configuration and $\#C_i\#$ denote the accepting configuration. We will have the following rules for each proper prefix $p$ (these are just strings of terminal characters) of $\#C_1\#$:

$$\text{NonStart} \rightarrow pz\text{ANY}$$

where $z$ is a character we are using to encode the computation history and $pz$ is not a prefix of $\#C_1\#$. We will have the following rules for each proper suffix $s$ (these are just strings of terminal characters) of $\#C_1\#$:

$$\text{NonAccept} \rightarrow \text{ANY}zs$$

where $z$ is a character we are using to encode the computation history and $zs$ is not a suffix of $\#C_1\#$.

Now a substring can fail to be a transition by being an improper encoding of a pair of configurations, or if the pair of configurations does not follow from the transition function. First, we must be able to have arbitrary strings on either side of the invalid transition:

$$\text{NonTransition} \rightarrow z\text{NonTransition} | \text{NonTransition}z$$

for $z$ is a character we are using to encode the computation history.

A full grammar is complicated but the idea can be understood from the following rules that show how a single variable on the right side can be used to generate valid transitions and some of the ways the transition can be determined to be invalid:

$$\text{NonTransition} \rightarrow \#\text{InvalidTransition}\#$$

An invalid transition is one that does not contain any states

$$\text{InvalidTransition} \rightarrow \text{InvalidTransitionWithoutState}|\text{ValidTransitionWithoutState}$$

A transition is invalid if it contains multiple states in either configuration

$$\text{InvalidTransition} \rightarrow q \text{InvalidTransitionWithState}|q \text{ValidTransitionWithState}\ |
\text{InvalidTransitionWithState}\ q | \text{ValidTransitionWithState}\ q$$

A valid transition must have a $\#$ in the middle and have symmetry up to a certain point (the tape head area).

$$\text{ValidTransitionWithoutState} \rightarrow \#|z \text{ValidTransitionWithoutState} z$$

If the transition $aqb \rightarrow acq'$ is valid according to the transition function

$$\text{ValidTransitionWithState} \rightarrow aqb \text{ValidTransitionWithoutState}\ q'ca|z \text{ValidTransitionWithState}\ z$$

$\text{InvalidTransitionWithState} \rightarrow aqb \text{InvalidTransitionWithoutState}\ q'ca|z \text{InvalidTransitionWithState}\ z$

If the transition $aqb \rightarrow acq'$ is not valid according to the transition function

$$\text{InvalidTransitionWithState} \rightarrow aqb \text{ValidTransitionWithoutState}\ q'ca$$

If the configurations do not match past the area around the tape head, the transition is invalid

$$\text{InvalidTransitionWithState} \rightarrow y \text{ValidTransitionWithState}\ z$$

Assume $D$ is a decider for $\text{ALL}_{\text{LIN}}$. The following TM $A$ is a decider for $A_{TM}$

$M = \text{"On input } \langle M, w \rangle$\n  \begin{itemize}
    \item Create the linear grammar $G$ as outlined above
    \item Run $D$ on $\langle G \rangle$
    \item If $D$ accepts, reject, and if $D$ rejects, accept.
  \end{itemize}
Thus we have a contradiction so $ALL_{LIN}$ is undecidable.

3) A simple answer is that since $SELFTM$ outputs its description on any input, an LBA cannot accomplish this since on an empty input, there is no space to output the description of the LBA.

However, if we assume that the LBA outputs its description when given a sufficiently large input, we have to justify using parts of the recursion theorem. First, and LBA is unable to compute a computable function in general. Though a more complex alphabet can be used to compute something like $f(w) = ww$ in the following way: $f(abc) = (a,a)(b,b)(c,c)$ where each $(x,y)$ is an alphabet symbol for every $x, y$ in the original alphabet of $w$ and this encoding is understood as first taking every first element of each ordered pair in turn, and then taking every second element of every ordered pair in turn. However, a function like $f(w) = w|w|$ cannot be computed since for any new alphabet, there is a sufficiently large $w$ so that this alphabet is unable to represent $w|w|$ within $|w|$ symbols. What is crucial is that the length of the description of the LBA that prints $w$ on any sufficiently large input is linear to the length of $w$. A naive construction is to make a state for each character in $w$, a transition from each of these states to the next that writes the corresponding letter and moves right, and then make a state for erasing the rest of the string. Certainly this is linear to $|w|$ so we could make an LBA that computes this function. The rest of the proof of the recursion theorem can be applied at this point so with the assumption that the input is large enough, an LBA can output its own description.

4) The proof is nearly identical to the proof for $MINTM$

Assume that some TM $E$ enumerates $NEARLYMINTM$ and obtain a contradiction. We construct the following TM $C$:

$C = \"On input $w$\$
- Obtain, via the recursion theorem, own description $\langle C \rangle$.
- Run the enumerator $E$ until a machine $D$ appears with description longer than the square of the length of $C$'s description.
- Simulate $D$ on input $w$ and decide the same way as $D$.

Since $NEARLYMINTM$ is infinite (infinitely many languages), $E$ must print a machine $D$ with length longer than the square of $C$'s description. Then $C$ behaves precisely as $D$ so they accept the same language but the description length of $C$ is less than the square root of the description length of $D$ so we have a contradiction. Thus, $NEARLYMINTM$ is unrecognizable.

5) This proof is also similar but uses different justification for finiteness

Assume that some TM $E$ enumerates $MINSTATE_{TM}$ and obtain a contradiction. We construct the following TM $C$:

$C = \"On input $w$\$
- Obtain, via the recursion theorem, own description $\langle C \rangle$.
- Run the enumerator $E$ until a machine $D$ appears with more states than $C$ and whose alphabet is at least as large as that of $C$
- Simulate $D$ on input $w$ and decide the same way as $D$.

let $k = |Q|$ and $j = |\Gamma|$. There are only finitely many machines with at most $k$ states and $j$ tape alphabet symbols—given $a$ states and $b$ alphabet symbols there are $a*b*2$ possible transition functions. Since $MINSTATE_{TM}$ is infinite (infinitely many languages) $E$ must print a machine $D$ more states and at least as many alphabet symbols as $C$ has so we have a contradiction. Thus, $MINSTATE_{TM}$ is unrecognizable.