# Harvard University
# Computer Science 121

## Problem Set 5

Due Friday, October 29, 2010 at 1:20 PM.
Submit a single PDF (lastname+ps5.pdf) of your solutions to cs121+ps5@seas.harvard.edu
Late problem sets may be turned in until Monday, November 1, 2010 at 1:20 PM with a 20% penalty.
See syllabus for collaboration policy.

## Name

Problem set by !!! Your Name Here !!!

with collaborators !!! Collaborators' names here !!!!

## Notes

Try this week's BONUS problem! We think you'll like it.
Use the provided template for your answers. Custom formatted solutions are difficult to grade.
Read instructions carefully. Don't forget one side of an IFF.

PROBLEM 1 (4 + 4 + 2 + 2 points)

Let $G = (V, \Sigma, R, S)$ where $V = \{S, V\}$, $\Sigma = \{a, b\}$, and $R$ is the set of rules:

$$S \to bSS \mid aS \mid aV$$

$$V \to aVb \mid bVa \mid VV \mid \varepsilon$$

(A) Transform $G$ into an equivalent grammar $G'$ in Chomsky normal form.

(B) Verify that the string $abaab$ is generated by $G'$, using the recognition algorithm for grammars in Chomsky normal form given in class. Show the complete filled-in matrix.

(C) Draw a parse tree for the derviation of $abaab$ from the transformed grammar $G'$.

(D) In one sentence, what language does $G$ generate?

(A)

Add a new start symbol:

$$S' \to S$$

$$S \to bSS \mid aS \mid aV$$

$$V \to aVb \mid bVa \mid VV \mid \varepsilon$$

Eliminate $V \to \varepsilon$:

$$S' \to S$$

$$S \to bSS \mid aS \mid aV \mid a$$

$$V \to aVb \mid bVa \mid VV \mid ab \mid ba$$

Eliminate long rules:

$$S' \to S$$

$$S \to bW \mid aS \mid aV \mid a$$

$$V \to aX \mid bY \mid VV \mid ab \mid ba$$

$$W \to SS$$

$$X \to Vb$$

$$Y \to Va$$

Eliminate unit rules:

$$S' \to bW \mid aS \mid aV \mid a$$

$$S \to bW \mid aS \mid aV \mid a$$

$$V \to aX \mid bY \mid VV \mid ab \mid ba$$

$$W \to SS$$

$$X \to Vb$$

$$Y \to Va$$

Eliminate terminal-generating rules:

$$S' \to BW \mid AS \mid AV \mid a$$

$$S \to BW \mid AS \mid AV \mid a$$

$$V \to AX \mid BY \mid VV \mid AB \mid BA$$

$$W \to SS$$

$$X \to VB$$

$$Y \to VA$$

$$A \to a$$

$$B \to b$$

(B)

$S'$ is present in the lower left corner, so $abaab$ is generated by $G'$.

```
     a
|S'SA|    b
|V   |B   |    a
|S'SY|V   |S'SA|    a
|S'SW|S'SY|S'SW|S'SA|    b
|S'S |V   |S'S |V   |B   |
```

(C)

```
        S'
      /    \
    A        V
    |      /   \
    |    V       V
    |   / \     / \
    |  B   A   A   B
    |  |   |   |   |
    a  b   a   a   b
```

(D) Strings with more a's than b's.

## PROBLEM 2 (6 points)

A left-linear rule of a CFG $(V, \Sigma, R, S)$ is one whose right-hand side is a member of $V\Sigma^* \cup \Sigma^*$, that is, only the leftmost symbol can be a nonterminal. Right-linear rules are defined analogously. Show that a grammar with only left-linear rules, or only right-linear rules, generates a regular language, but a grammar with a mixture of left- and right-linear rules may generate a non-regular language.

(A)
Consider a right-linear grammar. We construct a GNFA recognizing strings generated by this grammar by creating a state per variable and transitions between variables where if $V \to \mu V'$ is a transition we have a transition from V to V' with the string $\mu$. For transitions without a variable at right we transition to a new accept state (we only need one). We mark the state corresponding to S as the start state.

Left-linear grammars can be mapped to right-linear grammars. Any rule $V \to \mu V'$ is rewritten $V \to V'\mu^R$, and any rule $V \to \mu$ is rewritten $V \to \mu^R$. This right-linear grammar generates the reverse of the language recognized by the left-linear grammar, which is regular as shown above. Since regular languages are closed under reversal left-linear grammars also generate regular languages.

Here's a grammar with a mixture of rules generating $a^n b^n$, which we know is not regular:

$$S \to aB \mid Ab \mid \epsilon$$

$$B \to Sb$$

$$A \to aS$$

## PROBLEM 3 (6 points)

Consider two languages, $L_1$ and $L_2$, recognized by Turing machines $M_1$ and $M_2$ respectively. Prove there is a Turing machine, $M_{1+2}$ which recognizes $L_{1+2} = L_1 \cap L_2$. You may use the multitape Turing Machine model.

We'll use a multi-tape TM with two tapes in addition to the input tape for simplicitity. First we copy the input string to each tape. Second we simulate $M_1$ on the first tape. If it rejects we reject, if it accepts we simulate $M_2$ on the second tape and accept or reject as it does.

Suppose $w \in L_1 \cap L_2$. Then $M_1$ and $M_2$ must accept it, so $M_{1+2}$ does by construction, too, so $L_1 \cap L_2 \in M_{1+2}$. If $w \notin L_1 \cap L_2$, one machine must reject or loop forever. If either rejects so does $M_{1+2}$ and if either loops forever so does $M_{1+2}$, so $L_1 \cap L_2 = M_{1+2}$.

PROBLEM 4 (6 + 2 points)

Consider a Turing machine $M$ with a tape alphabet of size $n \geq 2$ deciding a language $L$.

(A) Show how to construct a Turing machine with a two-symbol tape alphabet that decides $L$.

(B) Compare your constructed machine's and $M$'s time to halt. That is, if $M$ takes $N$ steps to halt on input $w$, approximately how many steps, as a function of $N$, $n$, and perhaps other parameters of $M$, will your transformed machine take?

(A)

We consider an alphabet over $\{\sqcup, 0\}$ to fit convention. For any set of symbols $\Delta$ of finite size $m$ we can contruct a unary fixed width encoding by numbering each symbol $1, 2 \ldots m$ and mapping them to strings $0^i \sqcup^{m-i}$ where $i$ is the symbol's value in our numbering.

We know need to show how to implement the following three functions:

**Move left** Move left $2m - 1$ on the tape. Translate the symbol.

**Move right** Move right 1 space. Translate the symbol.

**Write symbol** Move $m - 1$ to the left. Write the symbol's translation.

The above works since after reading each symbol we're on the last space occupied by that symbol. Moves to the left or right simply put the head on the start of the next symbol relative to this point.

We haven't described the translation process. For each symbol we create a finite set of states which permit us to count the number of zeros in a fixed width segment. We then map this count into the original symbol for the Turing machine's original logic to reason over.

If we didn't use a fixed width encoding (which is fine) each write may have required rewriting the entire string to the right.

(B)

As a worst case we take rougly $< 4mN$ steps to halt since for each movement of the original machine we may now perform $4m$ moves in our construction (move left, $2m$, $m$ read, $m write$).

PROBLEM 5 (6 + 2 points)

A queue is similar to a stack, except that instead of pushing and popping on the top of the stack, you enqueue an item on one end of the queue and dequeue it on the other end in a first-in/first-out manner.

A queue automaton (QA) is a deterministic automaton that, in addition to having a finite set of states and transitions, has a queue for data storage. At each step, a QA dequeues the next symbol, and based on that symbol and its current state, transitions to another state and queues any number of symbols . When run on an input string $w$, a QA begins with the string $w\$$ in the queue, where \$ is a symbol not in the input alphabet that marks the end of the string. A QA accepts by transitioning to a special accept state. A QA can fail to accept by looping forever or if its queue becomes empty (so that it cannot dequeue a symbol and make any more transitions).

(A) Demonstrate how a Turing machine can be simulated by a QA. You need not prove your construction correct, but it should handle corner cases correctly.

(B) Estimate the number of states in your construction of a QA as a function of the number of states in the TM and the size of its alphabet.

(A)

As before it's sufficient to simulate moving the TM's head left or right, optionally replacing a character as it goes.

Before we start, however, let's describe how we'll associate the queue with the original machine. We'll copy over all the states from the TM and broadly we're considering the character dequeued at each step to be the character the head of the TM would read at that step. However, we'll have to transition through the queue sometimes to find the right character, so we'll be constructing new states to perform this action. In particular, we'll be constructing a set of states for "sliding" (described below) which shift the queue until a special marker, #, is found. These states will also remember the last character seen, so for each state in the original TM. $n$, we'll be constructing $m$, the size of the TM's alphabet + \$, new states. Well also need a few more special states for when we encounter the \$ symbol which will be well defined from context.

To avoid confusion, let's now discuss the queue itself, which contains the string read left to right, exactly as it would be on the tape, with a dollar sign at the end (as specified in the problem). The following two pictures show this:

$$\leftarrow \textit{dequeue}\ w\$ \leftarrow\ \textit{enqueue}$$

$$\text{end of string towards start}\ \leftarrow\ \$\ \rightarrow\ \text{start of string towards end}$$

Now we can discuss moving the TM's head and writing.

**Move right** Dequeuing the next symbol provides the symbol to be read. The symbol to write is enqueued. If the dequeued symbol is a \$ we're at the end of the input string. We enqueue $\sqcup\#\$$ and "slide" (explained later) along the string until we reach the $\sqcup\#$, at which point we act as if the TM saw the blank only at that step.

**Move left** The symbol to write, $w$, is enqueued $\#w$. We "slide" along the queue until we find the character proceeding the #. If it's a \$ we perform another dequeue / enqueue, shifting the queue so we're back at the first character of the tape (since we tried to run off the edge). If it's any other character we act as if the TM saw that character in its next step.

"Sliding" simply means the TM shifts the queue, enqueuing / dequeuing, but remembering the previous character seen. Since the number of characters is finite we may create states to do so.

We need a set of these for each state in the original since we need to remember which state in the original TM to transition back to once we've arrived at the correct character in the queue.

(B)

The above requires roughly $mn$ states following closely from the construction to handle sliding, plus some constant number of states for a few special cases.

Therefore, $Q$ will need a cross product of states $(\Gamma \cup S) \times (\Gamma \cup S)$, plus a few extra states to perform the initialization and to handle the first two symbols popped on each round, when there are not yet 2 previous symbols to be remembered. These extra states give only a small constant or linear overhead, so we can ignore these for the estimate.

So, we need on the order of $(|\Gamma| + |S|)^2$ states in our QA in total.

## PROBLEM 6 (+1 BONUS points)

Prove the language of even palindromes, $L = \{ww^R \mid w \in \Sigma^*\}$ is not recognized by any deterministic pushdown automata.

We can derive this from the formal properties of the DPDA and the language.

Assume there is a DPDA recognizing palindromes.

Any device recognizing palindromes must test each character at position $i$ against only one other at position $m-i$, where $m$ is the length of the string. By contradiction, assume there is such a device which does otherwise. If it does not test a character against another then there is some character we can arbitrarily change in a string such that it's no longer a palindrome. If it tests a particular character against more than one other we can't construct every palindrome. If it tests against a character at the wrong place we also do not recognize all or only all palindromes.

Since we're considering a DPDA, it must decide when to test a particular character based on its past history exclusively and it only receives one opportunity to do so since it only has one action per read character. Testing a character implies the string is of a particular length.

Consider some palindrome $p$. Palindrome $pp^R$ also must be accepted by definition. By the above our DPDA must test the first character of $p$ against the first character in $p^R$. It decides to do so after witness $p$. However, this means it makes no comparisons when reviewing $p$. A contradiction.