CS 341 Homework 20
Unrestricted Grammars

1. Find grammars that generate the following languages:
(a) L = {ww : w ∈ {a, b}*}
(b) L = { a^{2^n} : n ≥ 0}
(c) L = { $a^n b^{2n} c^{3n}$: n ≥ 1}
(d) L = {w^R : w is the social security number of a living American citizen}
(e) L = {$wc^m d^n$: w ∈ {a, b}* and m = the number of a's in w and n equals the number of b's in w}

2. Find a grammar that computes the function f(w) = ww, where w ∈ {a, b}*.

Solutions

1. (a) L = {ww : w ∈ {a, b}*}
There isn't any way to generate the two w's in the correct order. Suppose we try. Then we could get aSa. Suppose we want b next. Then we need Sa to become bSab, since the new b has to come after the a that's already there. That could work. Now we have abSab. Let's say we want a next. Now Sab has to become aSaba. The problem is that, as the length of the string grows, so does the number of rules we'll need to cope with all the patterns we could have to replace. In a finite number of rules, we can't deal with replacing S (which we need to do to get the next character in the first occurrence of w), and adding a new character that is arbitrarily far away from S.

The other approach we could try would be have a rule S → WW, and then let W generate a string of a's and b's. But this won't work, since we have no way to control the expansion of the two W's so that they produce the same thing.

So what we need to do is to generate ww^R and then, carefully, reverse the order of the characters in w^R. What we'll do is to start by erecting a wall (#) at the right end of the string. Then we'll generate ww^R. Then, in a second phase, we'll take the characters in the second w and, one at a time, starting with the leftmost, move it right and then move it past the wall. At each step, we move each character up to the wall and then just over it, but we don't reverse characters once they get over the wall. The first part of the grammar, which will generate wTw^R, looks like this:

$S \rightarrow S_1 \#$	This inserts the wall at the right.
$S_1 \rightarrow aS_1a$	
$S_1 \rightarrow bS_1b$	
$S_1 \rightarrow T$	T will mark the left edge of the portion that needs to be reversed.

At this point, we can generate strings such as abbbTbbba#. What we need to do now is to reverse the string of a's and b's that is between T and #. To do that, we let T spin off a marker Q, which we can pass rightward through the string. As it moves to the right, it will take the first a or b it finds with it. It does this by swapping the character it is carrying (the one just to the right of it) with the next one to the right. It also moves itself one square to the right. The four rules marked with * accomplish this. When Q's character gets to the # (the rules marked **), the a or b will swap places with the # (thus hopping the fence) and the Q will go away. We can keep doing this until all the a's and b's are behind the fence and in the right order. Then the final T# will drop out. Here are the rules for this phase:

$T \rightarrow TQ$
$Qaa \rightarrow aQa$ *
$Qab \rightarrow bQa$ *
$Qbb \rightarrow bQb$ *
$Qba \rightarrow aQb$ *
$Qa\# \rightarrow \#a$ **
$Qb\# \rightarrow \#b$ **
$T\# \rightarrow \varepsilon$
So with R as given above, the grammar G = ({S, S_1, #, T, Q, a, b}, {a, b}, R, S}

(b) L = { a^{2^n} : n ≥ 0}
The idea here is first to generate the first string, which is just a. Then think about the next one. You can derive it by taking the previous one, and, for every a, write two a's. So we get aa. Now to get the third one, we do the same thing. Each of the two a's becomes two and we have four, and so forth. So we need a rule to get us started and to indicate the possibility of duplication. Then we need rules to actually do the duplication. To make duplication happen, we need a symbol that gets generated by S indicating the option to repeat. We'll use P. Since duplication can happen an arbitrary number of times, we need P to spin off as many individual duplication commands as we want. We'll use R for that. The one other thing we need is to make sure, if we start a duplication step, that we finish it. In other words, suppose we currently have aaaa. If we start duplicating the a's, we must duplicate all of them. Otherwise, we might end up with, for example, seven a's. So we'll introduce a left edge marker, #. Once we fire up a duplication (by creating an R), we'll only stop (i.e., get rid of R) when R has made it all the way to the other end of the string (namely the left end since it starts at the right). So we get the following rules:

$S \rightarrow \#aP$	P lets us start up duplication processes as often as we like.
$P \rightarrow \varepsilon$	When we've done as many as we want, we get rid of P.
$P \rightarrow RP$	R will actually do a duplication by moving leftward, duplicating every a it sees.
$aR \rightarrow Raa$	Actually duplicates one a, and moves R one square to the left so it moves on to the next a
$\#R \rightarrow \#$	Get rid of R once it's made it all the way to the left
$\# \rightarrow \varepsilon$	Get of # at the end

So with R as given above, the grammar G = ({S, P, R, #, a, b}, {a, b}, R, S}

(c) L = { $a^n b^{2n} c^{3n}$: n ≥ 1}
This one is very similar to $a^n b^n c^n$. The only difference is that we will churn out b's in pairs and c's in triples each time we expand S. So we get:
$S \rightarrow aBSccc$
$S \rightarrow aBccc$
$Ba \rightarrow aB$
$Bc \rightarrow bbc$
$Bb \rightarrow bbb$
So with R as given above, the grammar G = ({S, B, a, b, c}, {a, b, c}, R, S}

(d) L = {w^R : w is the social security number of a living American citizen}
This one is regular. There is a finite number of such social security numbers. So we need one rule for each number. Each rule is of the form S → <valid number>. So with that collection of rules as R, the grammar G = ({S, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, R, S}

(e) L = {$wc^m d^n$: w ∈ {a, b}* and m = the number of a's in w and n equals the number of b's in w}
The idea here is to generate a c every time we generate an a and to generate a d every time we generate a b. We'll do this by generating the nonterminals C and D, which we will use to generate c's and d's once everything is in the right place. Once we've finished generating all the a's and b's we want, the next thing we need to do is to get

all the D's to the far right of the string, all the C's next, and then have the a's and b's left alone at the left. We guarantee that everything must line up that way by making sure that C can't become c and D can't become d unless things are right. To do this, we require that D can only become d if it's all the way to the right (i.e., it's followed by #) or it's got a d to its right. Similarly with C. We can do this with the following rules;

$S \rightarrow S_1\#$
$S_1 \rightarrow aS_1C$
$S_1 \rightarrow bS_1D$
$S_1 \rightarrow \varepsilon$
$DC \rightarrow CD$
$D\# \rightarrow d$
$Dd \rightarrow dd$
$C\# \rightarrow c$
$Cd \rightarrow cd$
$Cc \rightarrow cc$
$\# \rightarrow \varepsilon$
So with R as given above, the grammar $G = (\{S, S_1, C, D, \#, a, b, c, d\}, \{a, b, c, d\}, R, S\}$

2. We need to find a grammar that computes the function f(w) = ww. So we'll get inputs such as SabaS. Think of the grammar we'll build as a procedure, which will work as described below. At any given time, the string that has just been derived will be composed of the following regions:

<the part of w that has already been copied>	S	<the part of w that has not yet been copied, which may have within it a character (preceded by #) that is currently being copied by being moved through the region>	T (inserted when the first character moves into the copy region)	<the part of the second w that has been copied so far, which may have within it a character (preceded by %) that is currently being moved through the region>	W (also when T is)

Most of the rules come in pairs, one dealing with an a, the other with b.

$SS \rightarrow \varepsilon$	Handles the empty string.
$Sa \rightarrow aS\#a$	Move S past the first a to indicate that it has already been copied. Then start copying it by introducing a new a, preceded by the special marker #, which we'll use to push the new a to the right end of the string.
$Sb \rightarrow bS\#b$	Same for copying b.
$\#aa \rightarrow a\#a$	Move the a we're copying past the next character if it's an a.
$\#ab \rightarrow b\#a$	Move the a we're copying past the next character if it's a b.
$\#ba \rightarrow a\#b$	Same two rules for pushing b.
$\#bb \rightarrow b\#b$	"
$\#aS \rightarrow \#aTW$	We've gotten to the end of w. This is the first character to be copied, so the initial S is at the end of w. We need to create a boundary between w and the copied w. T will be that boundary. We also need to create a boundary for the end of the copied w. W will be that boundary. T and W are adjacent at this point because we haven't copied any characters into the copy region yet.
$\#bS \rightarrow \#aTW$	Same if we get to the end of w pushing b.
$\#aT \rightarrow T\%a$	Jump the a we're copying into the copy region (i.e., to the right of T). Get rid of #, since we're done with it. Introduce %, which we'll use to push the copied a through the copy region.
$\#bT \rightarrow T\%b$	Same if we're pushing b.

%aa → a%a	Push a to the right through the copied region in exactly the same way we pushed it through w, except we're using % rather than # as the pusher. This rule pushes a past a.
%ab → b%a	Pushes a past b.
%ba → a%b	Same two rules for pushing b.
%bb → b%b	"
%aW → aW	We've pushed an a all the way to the right boundary, so get rid of %, the pusher.
%bW → bW	Same for a pushed b.
ST → ε	All the characters from w have been copied, so they're all to the left of S, which causes S to be adjacent to the middle marker T. We can now get rid of our special walls. Here we get rid of S and T.
W → ε	Gid rid of W. Note that if we do this before we should, there's no way to get rid of %, so any derivation path that does this will fail to produce a string in {a, b}*.

So with R as given above, the grammar G = ({S, T, W, #, %,a, b}, {a, b}, R, S}