SOLUTIONS HW#5

1. Start with only one initial symbol ⊥ on the stack. Scan the expression from left to right:

(i) If the symbol is a (, push it onto the stack.
(ii) If the symbol is an operator, either unary or binary, push it onto the stack.
(iii) If the symbol is a constant, push a pointer to it onto the stack
(iv) If the symbol is a variable, and on top of the stack is an operator, then look it up in the symbol table and push a pointer to the symbol table entry onto the stack. If the variable has never been seen before, a new symbol table entry is created. If on top of the stack is a variable, then push the operator "?" onto the stack first and then do the same operation as above for the variable
(v) If the symbol is a), do a *reduce*. A reduce step consists of the following sequence of actions:
  (a) Allocate a block of storage for a new node in the expression tree. The block has space for the name of an operator and pointers to left and right operands
  (b) Pop the top object off the stack. If it is a pointer to an operand, save the pointer in the newly allocated node as the right operand. If not, go to c)
  (c) Pop the top object off the stack. It should be an operator. If so, save the operator name in the newly allocated node.
  (d) Pop the object off the stack. It should be a pointer to an operand. If so, save the pointer in the newly allocated node as the left operand
  (e) Pop the object off the stack. It should be a (. If not, give error
  (f) Push a pointer to the newly allocated node onto the stack

Now, if we want to use the precedence relation of the operators, we can modify this algorithm to handle expressions that are not fully parenthesized. Whenever we are about to scan a binary operator, we check to make sure there is an operand on top of the stack, then we look at the stack symbol immediately below it. If it is a symbol of lower precedence we push the first one, if it is of higher or equal precedence, then we reduce and repeat the process. For example, if you are about to scan a +, then pop off the top operand and operator. If the popped operator is a +, then push it and the operand back on and keep going. If the popped operator is a "·", then do a reduce on the "·" and its two operands, then recursively check the next operator on the stack with the same procedure.

If we use this algorithm to our problem we get the tree in the attached picture.

2. (a)

0			
A	1		
∅	A	2	
B, S	B, S	B	3

(b) $S \to ABS \mid AB$

A→CA | a
C→a
B→DA
D→b

0						
A, C	1					
A	A, C	2				
∅	∅	D	3			
S	S	B	A, C	4		
S	S	B	A	A, C	5	
∅	∅	∅	∅	∅	D	6

0						
A, C	1					
∅	D	2				
S	B	A, C	3			
S	B	A	A, C	4		
∅	∅	∅	∅	D	5	
S	∅	S	S	B	A, C	6

3.

1. scan until first ? and replace it with $\bar{a}$
2. scan back until you find ⊥
3. start scanning until you find the first *a* and replace it with $\underline{a}$; we need to double the number of $\bar{a}$’s at the end of the tape
    - 1’. scan right until first $\bar{a}$ found
    - 2’. replace $\bar{a}$ with *b* and add a *c* at the end
    - 3’. Scan left until $\bar{a}$ found. Go to 2’.
    - 4’. When no $\bar{a}$’s left, change all *b*’s and *c*’s with $\bar{a}$
4. go to 2

When no more *a*’s, we have on the tape $\underline{a}^m\bar{a}^{2^m}$. In order to get a^{2^m} we replace all $\bar{a}$ with *a* and for each $\underline{a}$, we replace it by *a* and we replace one *a* at the end by ?.

4.

1. If nothing after #, delete # and stop, otherwise scan right to blank, then move left by one to an a and replace with a'. (If n=0, then return m, else mark the end of the string.)
2. Scan left to left endmarker, then move right by 1. (Initialize left)
3. Loop: (Interchange m and n: $a^m\#a^n$ becomes $a^n\#a^m$ by moving the #)

```
   3.1 If this is #, replace with a, then scan right to a' and replace with #, then exit loop, else if this is an a, replace with a'', then scan right to a'.
   3.2 Replace a' with a, then move left.
   3.3 If this is #, then replace with a, then scan left to a'', replace with # and exit loop, else this is a, so replace with a'.
   3.4 Scan left to a'', replace with a and move right.
   3.4 Continue loop at 3.1

(Each pass through the loop below decreases m by n.  If the loop ends before n steps, then the remaining part of m is restored below to give m mod n)

5. Loop:
   (Initialize left)
   5.1. Scan left to left endmarker, then move right by one and replace by a'.
   5.2. Loop:
         5.2.1. Scan right to blank, then move left. If this is an a, replace with blank. Otherwise this is #, so go to step 6.  (we overshot and have to repair)
         5.2.2. Scan left to a'.  Replace with a and move right.
         5.2.3. If this is #, then scan left and continue loop at step 5.1 (we finished m-n), otherwise mark as a', move right, and continue loop at 5.2.1.

(Initialize left)
6. Scan left to left endmarker, then move right by one.
(This loop restores the remaining part of n to give m mod n by adding an a to the end of the tape for each a to the left of a')
7. Loop:
   7.1. If this is a', replace by a and continue loop at step 1, else this is a, so replace by a''.
   7.2. Scan right to blank and replace with a.
   7.3. Scan left to a''.  Replace with a and move right by 1.  Continue loop at step 7.1
```

5. If L1 and L2 are RE, then there must be TM's M1 and M2 which recognize L1 and L2: M1 halts on inputs which are in L1 and M2 halts on inputs which are in L2.

Intersection is easy: for an input string *w*, first run M1 on it. If it halts, run M2. String *w* is in L1 $\cap$ L2 if and only if it is in L1 and it is in L2. So M1 and M2 both halt on *w* if and only if *w* is in L1 $\cap$ L2.

Union is harder: *w* might lie in L2 but not L1. If we run M1 on *w*, it will run forever, never giving us a chance to run M2. The solution is to run M1 and M2 in "parallel" on *w*. We should alternate steps on each machine. If *w* is in L1 $\cup$ L2, then one of the machines halts. If that happens, we halt and accept *w*, even if the other machine has not halted yet.