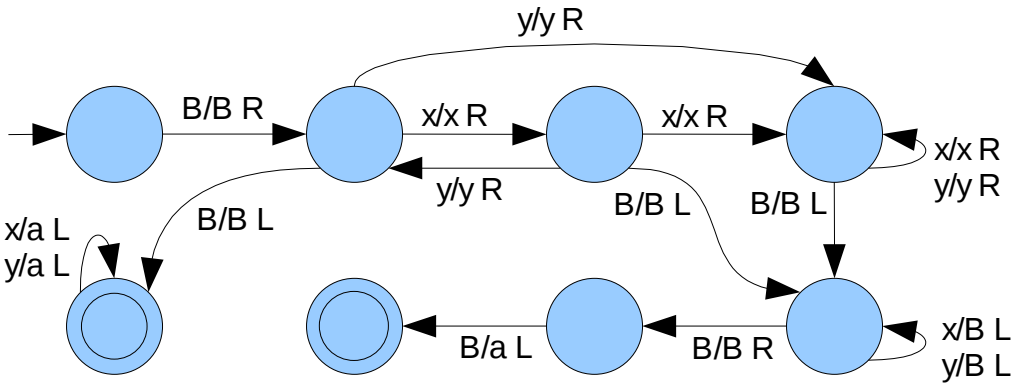
Formal Languages Midterm, 3rd Quarter

1.

(a)
I'll call the following representation Rf(M)
I use the same encoding as shown in 11.5, except for the following:
let f = qj `elem` F
where elem is the predicate testing set membership, returning boolean
if f = true then en(f) = 11 else en(f) = 1
The new encoding is:
en(qi)0en(x)0en(qj)0en(y)0en(d)0en(f)
For example:
(qj is not a final state)
d(q2, B) = [q1, 1, R]
111011101101101101
b)
I'll call this modified Universal machine 'Uf'
Everything is identical to the original Universal Turing Machine, except that the input is of the form Rf(M)w, and steps 4a and 4b are changed to:
4a. Tape 1 is scanned for a transition whose first two components match en(qi) and en(x). If there is no such transition, Uf moves to the right forever.
If tape 1 contains an encoded transition en(qi)0en(x)0en(qj)0en(y)0en(d)0en(f), then
i) en(qi) is replaced by en(qj) on tape 2
ii) The symbol y is written on tape 3
iii) The tape head of tape 3 is moved in the direction specified by d.
iv) If en(f) = 11, then U halts accepting the input.

2.

(a) This one nearly accepts the language on its own



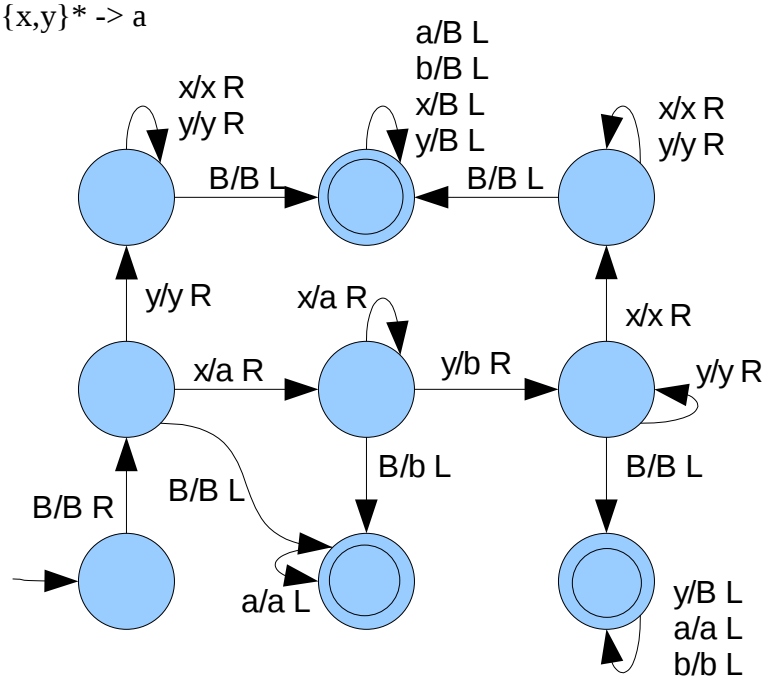
i) for strings of the form (xy)*, convert to (aa)*
ii) if the string starts with a y, ends with an x, contains two x's in a row, contains two y's in a row, erase the input, replacing it with an a

Example reductions:

w elem S1*	In L?	r(w) elem S2*	In Q?
lambda	yes	lambda	yes
xyxyxy	yes	aaaaaa	yes
xyxyx	no	a	no
xxxyyy	no	a	no

Generalized reductions:
(xy)* -> (aa)*
y+(xy)* | (xy)*x+ -> a
{x,y}*{xx | yy}{x,y}* -> a

(b)



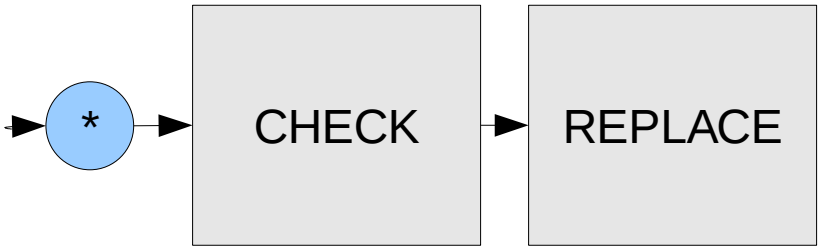
i) replace leading x's with a's followed by one b
ii) erase trailing y's
iii) if the string begins with y, has x's following y's, or is empty, then erase the input.

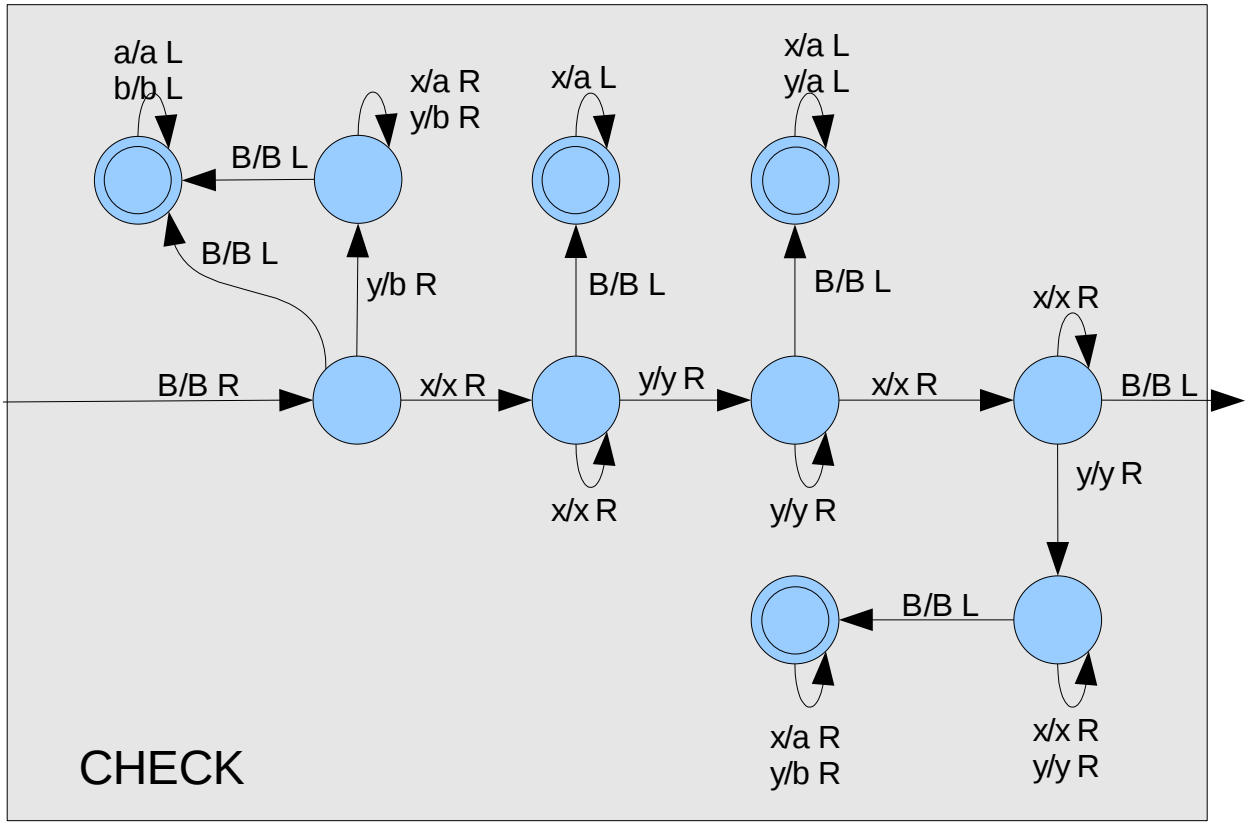
Example reductions:

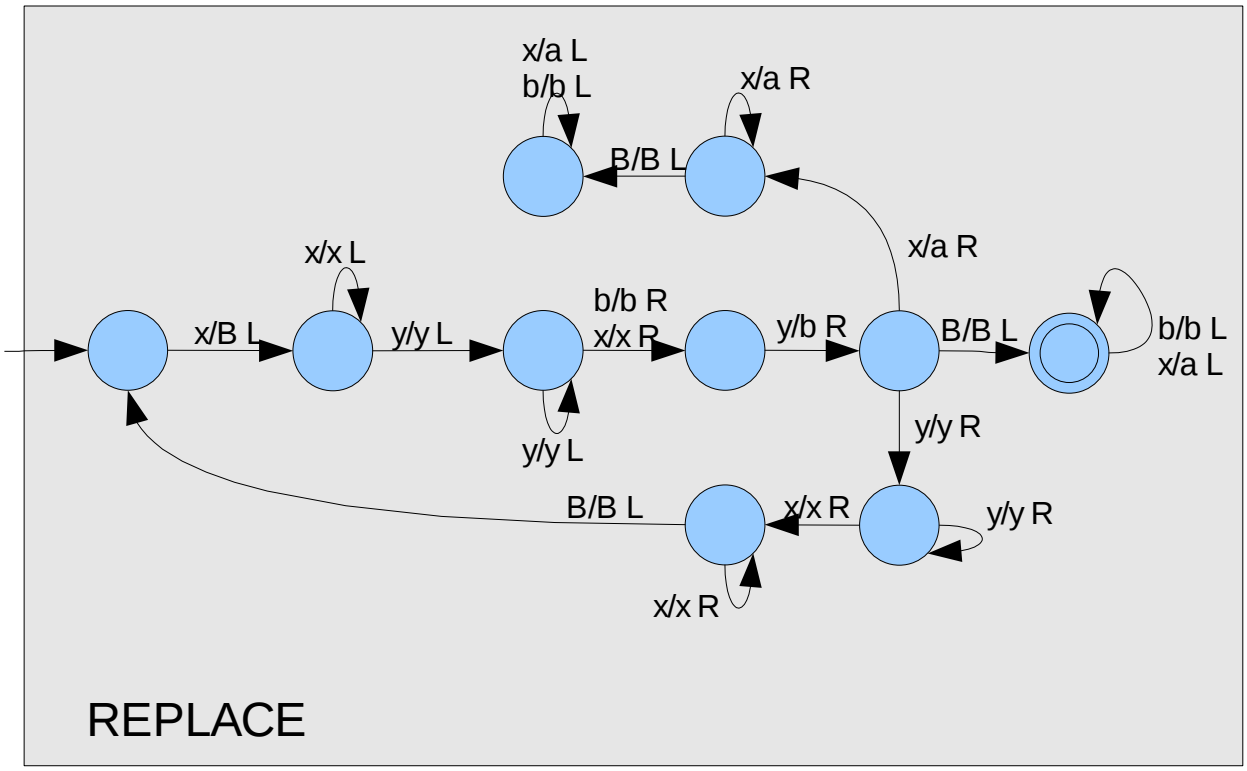
w elem S1*	in L?	r(w) elem S2*	In Q?
lambda	no	lambda	no
xxxx	yes	aaaab	yes
xxyyy	yes	aab	yes
yyxx	no	lambda	no
xxyx	no	lambda	no

Generalized reductions:
x+y* -> a+b
y*{x,y}* -> lambda
{x,y}*y+{x,y}*x+{x,y}* -> lambda (x's following y's)
lambda -> lambda

(c)







1. CHECK:
i) Check that the input string is of the form x+y+x+. If this format is not satisfied, produce an invalid string. This machines exits at the end of the string
2. REPLACE:
i) Starting at the end of the string, for each trailing x and y, erase the x and replace the y with a b. If all the x's are erased, then convert any remaining y's to a's (an invalid string), and move left, also replacing x's with a's. If all the y's are replaced by b's, then replace any remaining x's with a's, moving to the beginning.
Example reductions:

w elem S1*	**in L?**	**r(w) elem S2***	**In Q?**
lambda	yes	lambda	yes
xxxyyyxxx	yes	aaabbb	yes
xyyyx	no	abaa	no
xyxxx	no	abaa	no

Generalized reductions:
lambda -> lambda
$x^i \mid y^i \rightarrow a^i$
$x^i y^j \rightarrow a^{i+j}$
$y\{x,y\}^i \rightarrow b\{a,b\}^i$
$x^i y^j x^k y\{x,y\}^l \rightarrow a^i b^j a^k b\{a,b\}^l$
$x^i y^j x^k \rightarrow a^i b^j$ where j = k
$x^i y^j x^k \rightarrow a^i b^j a^{k-j}$ where j<k

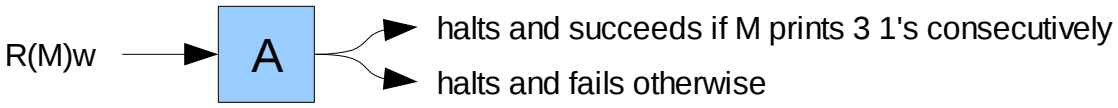
$x^iy^jx^k$ -> $a^ib^ka^{j-k}$

3.

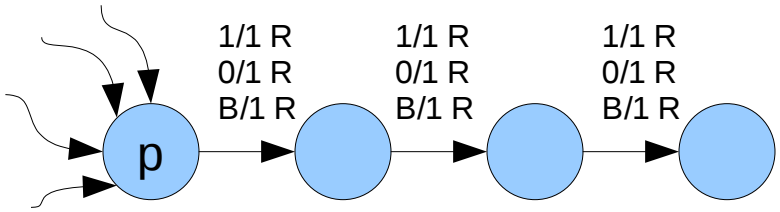
The 'halts on nth transition problem' is recursive because we've set an upper finite bound on the number of possible transitions. R(M)w0001^{n+1} will halt after n+1 transitions no matter what, so an infinite loop is not possible. The language of the halting problem is recursively enumerable but not recursive, as proved in ch. 12.

4.

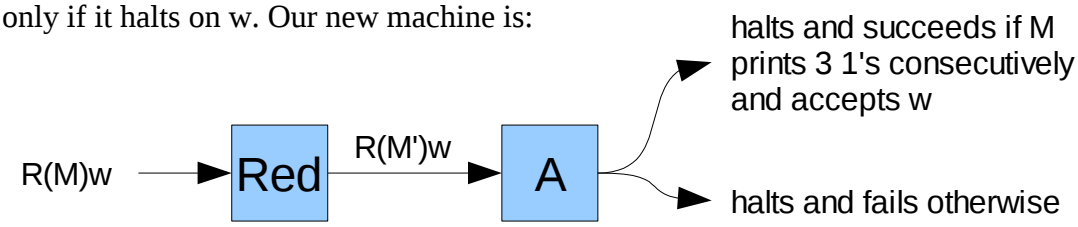
The halting problem can be reduced to this problem. Assume we have a machine A that takes as its input R(M)w and decides whether M prints three 1's consecutively:



Now let us define a reduction *Red* that takes as its input R(M)w:
1. Change the alphabet of M so that all 1's become Z's
2. Convert all transitions in M with 1's to transitions with Z's
3. Convert all instances of 1's in *w* to Z's
4. Take all undefined outgoing transitions in M and route them to a state P:



This creates a new machine called M'. Thus, R(M')w will print three 1's consecutively if and only if it halts on w. Our new machine is:



Thus, our machine A, which was assumed to exist, has been shown to solve the Halting Problem, which is impossible.

5.

I'll use indentation trees to show there are no solutions. Each node with n dominoes has successors each with n+1 dominoes, where the n+1st domino is a possible new move. Each branch will terminate with either an endless loop or will have no more possible moves. Also, each rule application is labeled 1-4 from left to right. I think the loops could be proved to be loops using induction, but I assume you aren't looking for that kind of rigor.

(a)

```
#1
[b ]
[ba]
     #1  #4
     [b ][ab]
     [ba][ba]
          #1  #4  #4*
          [b ][ab]([ab])^i  (or {b(ab)*}) infinite loop
          [ba][ba]([ba])^i  (or {ba(ba)*}) infinite loop
```

(b)

```
#1
[ab]
[a ] No options.
#2
[ba ]
[bab]
     #2   #3
     [ba ][b ]
     [bab][aa] No options.
     #2   #4
     [ba ][ba]
     [bab][ab]
          #2   #4  #3
          [ba ][ba][b ]
          [bab][ab][aa]  No options.
          #2   #4^i
          [ba ][ba]^i (or (ba)^n) - endless loop.
          [bab][ab]^i  (or b(ab)^n)
```

6.

S_U -> bS_U1 | b1

S_U -> aaS_U2 | aa2

S_U -> abS_U3 | ab3

S_L -> bbS_L1 | bb1

S_L -> $baaS_L2$ | baa2

S_L -> aS_L3 | a3

$L(G_U)$ intersected with $L(G_L)$ is not empty. abaa23 and abb13 are two example strings in both languages.

7.

I don't entirely understand this proof because I find it hard to read the Scheme, having no experience with the language, but I'll try to discuss some of the points.

- Instead of a TM that decides a language, we use a lambda function (lambda(x) ...), where x is the input string.
- Proof of the halting problem: assume there is a decider function A with input w. I can't decipher the definition of the halting problem function, but it looks like 'strange' will halt iff 'strange' loops on 'strange', which is the diagonalization
- Rice constructor: A1 and A2 are two acceptors. A2 is run on some new input x iff A1 accepts w.
- Rice's theorem: EM is the language of languages that have the property M, which are called L (I think). AL will decide if the input string x has the property defining L. The rice constructor is used to first run some accepter A on w, and will run AL on x iff A halts on w, which is the same thing as our book's proof.

8.

(a)

f(3,0) = g(3) = 3
f(3,1) = h(3,0,f(3,0))
= h(3,0,3)
= 3+3 = 6
f(3,2) = h(3,1,f(3,1))
= h(3,1,h(3,0,f(3,0)))
= h(3,1,h(3,0,3))
= h(3,1,6)
= 9

(b)

f = mult . (p_1^2, p_2^2 . s)
However, mult is itself recursive, and I can't think of a truly non-recursive definition

9.

(a) f = g . (p_1^2, p_2^2, p_1^2)
(b) f = g . (p_1^4, p_2^4, p_1^4)
(c) f = g . (c_1^1, c_2^1, p_1^1)