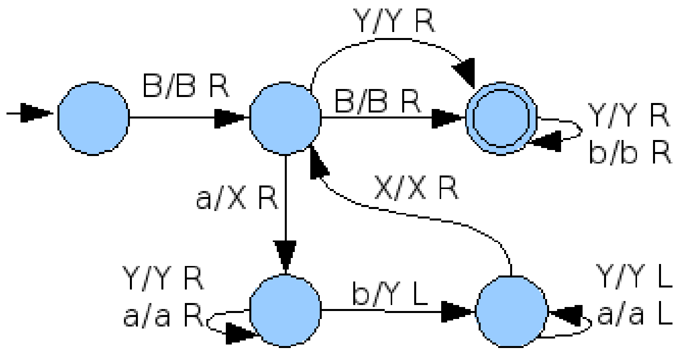
J Bolton March 15, 2011

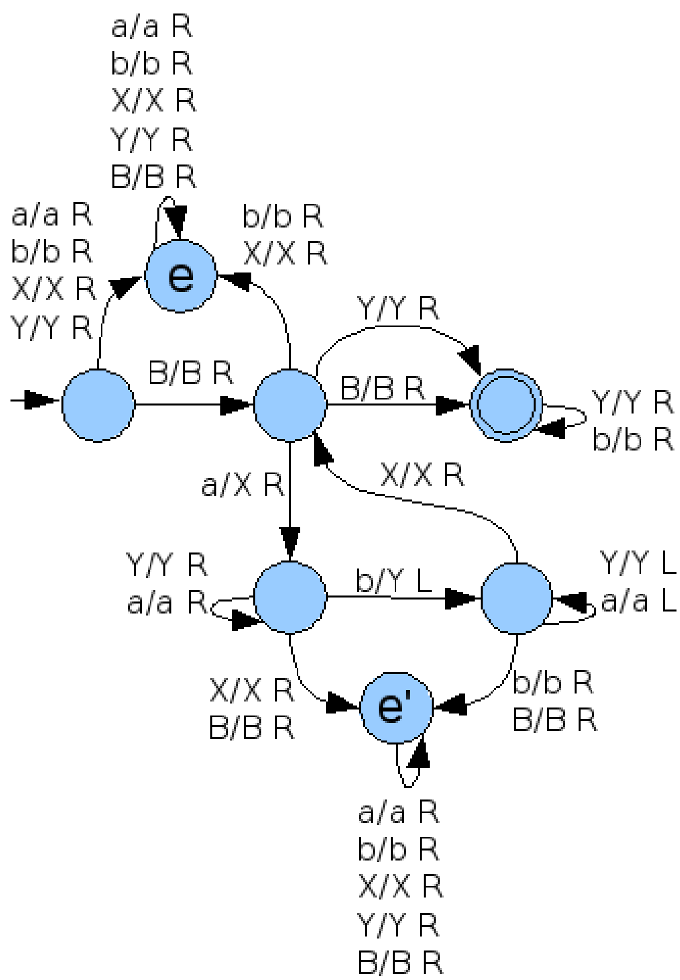
Logic/Formal Languages Final

2nd Quarter

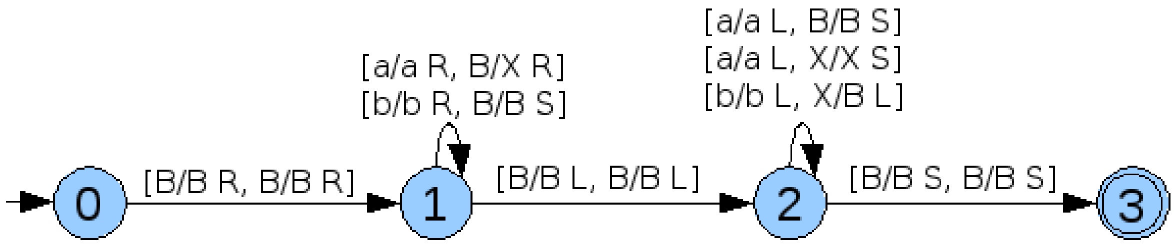
1.
This is basically a simplified version of the design in example 8.2.2



2.
Used theorem 8.3.2. I added two non-halting failure states instead of one to make the graph a little cleaner

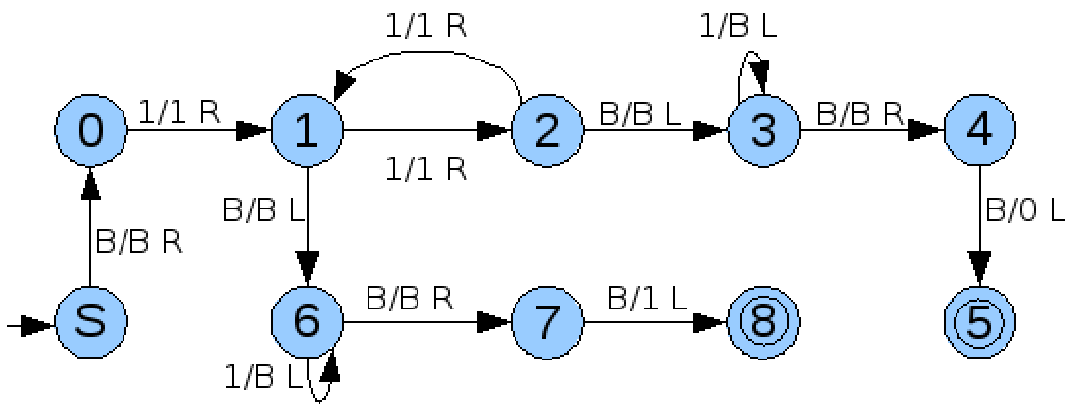


3.
This treats the bottom tape as a stack, sweeping from left to right pushing X's for a's, then sweeping right to left popping X's for b's



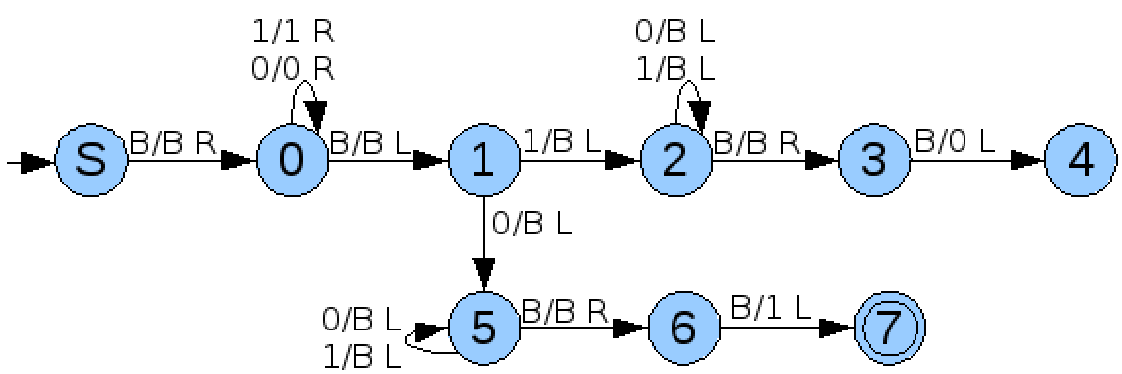
4.
I did (a) first for series of 1's, and then did another for binary after I realized it might be simpler, and I figured I might as well keep both.
(a) 1^{n+1} input (States 1,6-8 are the even case, 2-5 the odd)

qS*B111B*
*B**q0**111B*
*B1**q1**11B*
*B11**q2**1B*
*B111**q1**B*
*B11**q5**1B*
*B1**q5**1BB*
*B**q5**1BBB*
q5*BBBBB*
*B**q6**BBBB*
q7*B1BBB*
accepts.

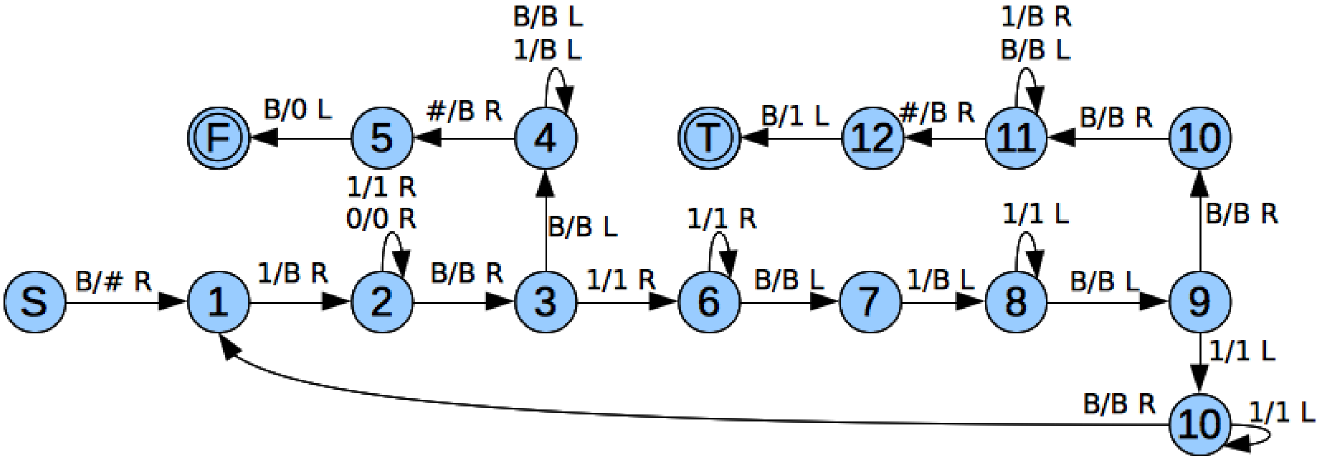


(a') Binary input (2-4 is the odd case, 5-6 the even case)

qS*B1010B*
*B**q0**1010B*
*B1**q0**010B*
*B10**q0**10B*
*B101**q0**0B*
*B1010**q0**B*
*B101**q1**0B*
*B10**q6**1BB*
*B1**q6**0BBB*
*B**q6**1BBBB*
q6*BBBBBB*
*B**q7**BBBBB*
q8*B1BBBB*



(b) Uses 1^{n+1} input. This alternately erases 1's from the outside of each parameter until one parameter is blank. It marks the beginning as '#'. States 4,5,F are entered when we know the first param is greater. States 10,11,12,T are entered when we know the first param is less.

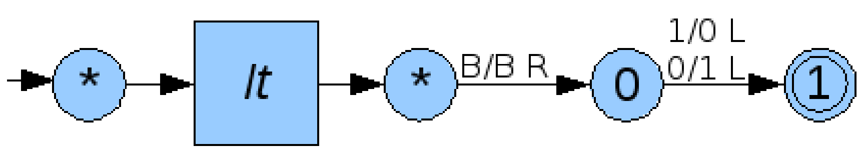


***qS**S1B11B*
*#**q1**1B11B*
*#B**q2**B11B*
*#BB**q3**11B*
*#BB1**q6**1B*
*#BB11**q6**B*
*#BB1**q7**1B*
*#BB**q8**1BB*
*#B**q8**B1BB*
*#**q9**BB1BB*
*#B**q10**B1B*
*#BB**q11**1B*
*#BBB**q11**B*
*#BB**q11**BB*
*#B**q11**BBB*
*#**q11**BBBB*
***q11**#BBBB*
*B**q12**BBBB*
***T**B1BBBBB*

5.

The macro *lt* is 4b's machine.

ltB11B1111B
ltB1BBBBBBB
B**q0**1BBBBBBB
q1B0BBBBBBB



6.

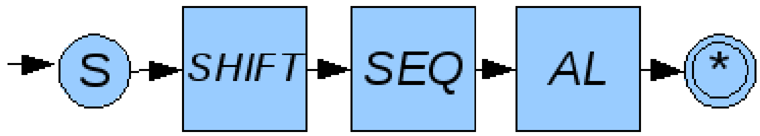
(a) n=0, m=4	(b) n=1,m=0	(c) n=2,m=2
q0B1B11111B	**q0**B11B1B	**q0**B111B111B
B**q1**1B11111B	B**q1**11B1B	B**q1**111B111B
B1**q2**B11111B	B1**q2**1B1B	B1**q2**11B111B
B1B**E1**11111B	B1X**q3**B1B	B1X**q3**1B111B
B1B**E1**BBBBBB	B1XB**CPY1**1B	B1X1B**CPY1**111B
B1B**ML1**BBBBBB	B1**q5**XB1B1B	B1X**q5**1B111B111B
*B1BBBBBBB	B**q4**1BB1B1B	B1**q6**X1B111B111B
	q4BBBB1B1B	B1X**q7**1B111B111B
	B**T**BBB1B1B	B1XX**q8**B111B111B
	B**E1**1BBBB1B	B1XXB**CPY11**111B111B
	B**T**BBBBB1B	B1XXB**MR1**111B111B111B
	B*1BBBBBBB	B1XXB111**A**B111B111B
		B1XXB111**ML1**B11111B
		B1XX*B111B11111B
		B1X**q5**XB111B11111B
		B1**q4**XBB111B11111B
		B**q4**1BBB111B11111B
		q4BBBBB111B11111B
		B**T**BBBB111B11111B
		B**E1**111BBBBB11111B
		B**T**BBBBBBBB11111B
		B*11111BBBBBBBB

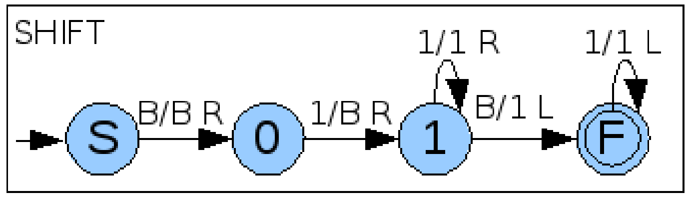
Isn't B/B R in the first transition of T, E, and CPY? Why would MULT do that transition before entering those macros?

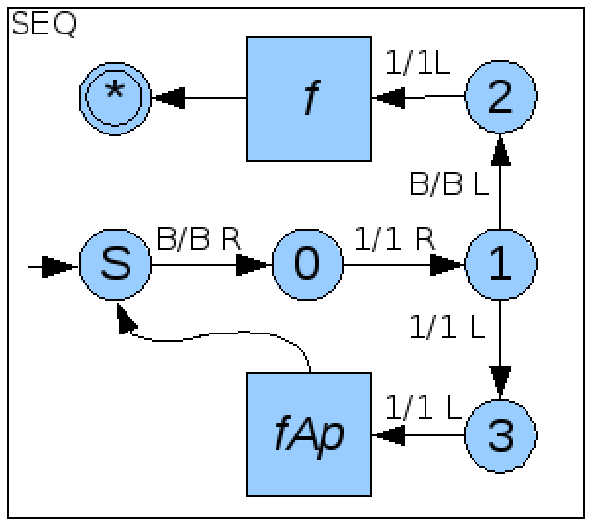
7.

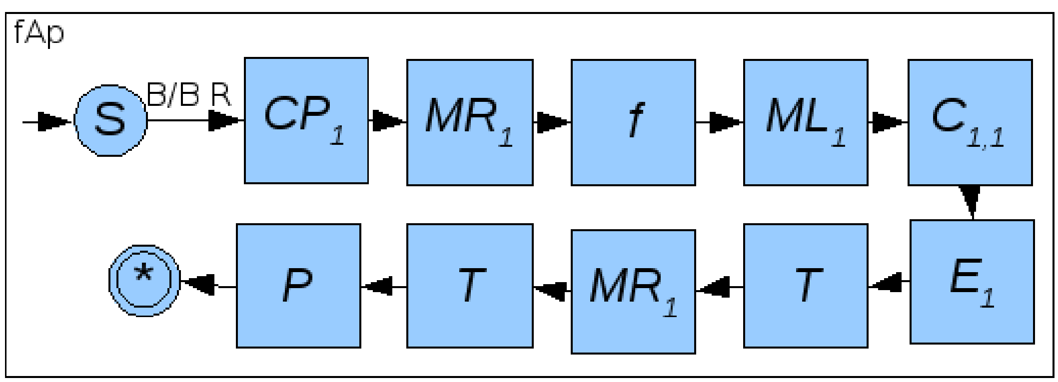
Design points:

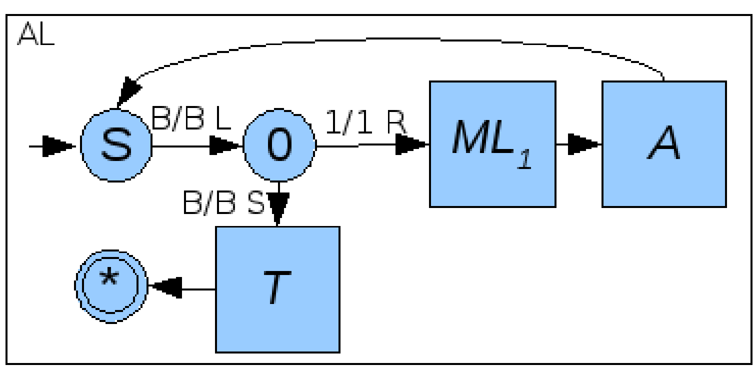
- P is predecessor from example 9.2.2. A is addition defined in 9.2.1
- This machine applies the function like: A(f(n), A(f(n-1), ... A(f(1), f(0))))
- fAp is “f Apply,” AL is “Add Left,” and SEQ is “Sequence”
- SEQ duplicates its parameter, applies f(n) to the duplicate, then moves the original past the duplicate and subtracts one from it. The reason for this sequence is that f(n) must be applied to a parameter with unlimited blanks after it, because it is possible that f(n) > n
- SEQ will leave a series of f(n), f(n-1) ... f(0) on the tape, and will stop at the beginning of the rightmost number.
- AL will repeatedly test for a parameter to the left and add going backwards.
- SHIFT shifts the first parameter to the right one place, making two blanks in the beginning so that AL will know when to stop.


S
SHIFT
SEQ
AL
*


SHIFT
S
B/B R
0
1/B R
1
1/1 R
B/1 L
F
1/1 L


SEQ
*
f
1/1L
2
B/B L
S
B/B R
0
1/1 R
1
1/1 L
fAp
1/1 L
3

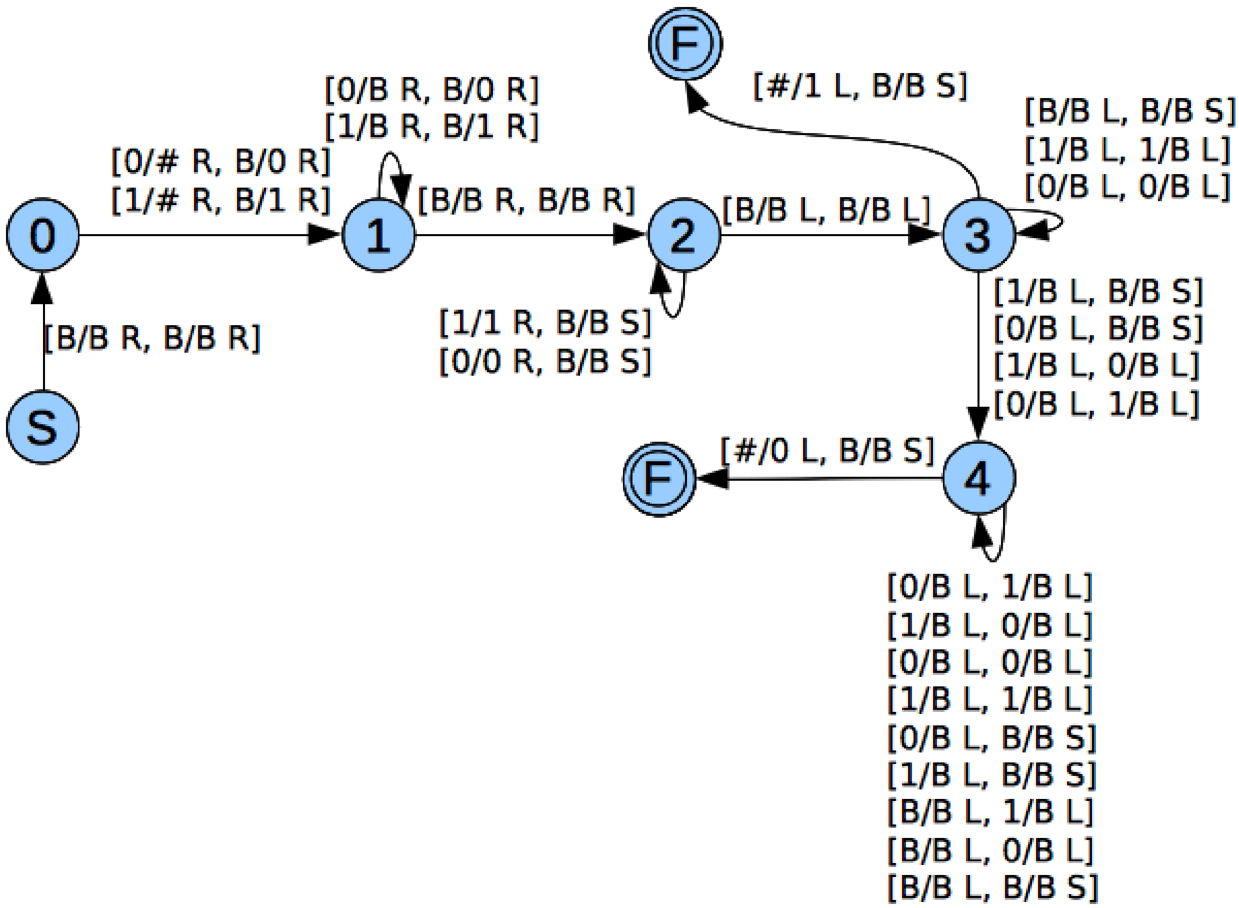

fAp
S
B/B R
CP1
MR1
f
ML1
C1,1
E1
T
MR1
T
P
*


AL
S
B/B L
0
1/1 R
ML1
A
B/B S
T
*

8.
(a). S => SBA => SAB => aAB => aaBB => aabB => aabb
(b). {aabb, a}
(c). S -> aabb | a

9.
(a).
S -> aXbba | abba
X -> aXbbA | abbA
Ab -> bA
Aa ->aa
(b).
S => aXbba
=> aabbAbba
=> aabbbAba
=>aabbbbAa
=> aabbbbaa
(c).
It would be similar to the design of the machine for $a^ib^ic^i$ in example 8.2.2. It would have a cycle that marked a left 'a' as an X, two middle b's as Y's, and a right 'a' as an X. As soon as all the left a's are marked as X's, then the head moves right over everything, only reading X's and Y's, until it reaches a blank, which would be the final state.

10.
Sweeps to the right, pushing the first parameter to the second tape, goes to the end of the tape, and works left, matching the second parameter to the first. This will take 4(length(u)+1) transitions assuming length(u) = length(v). I'm very curious how this could be made to take 3(length(u)+1) transitions.
(see next page).



11.

Due to time constraints, and since I have already spent 15+ hours on the turing machines for this test already, I will describe the design for this one:

- Mark the beginning as #
- First check for three leading zeros. If not, then erase input and return a 0.
- Check that the first state encoding is a 1. If not, erase input and return a 0.
- Create an OR macro that applies the Or logical operator to any amount of parameters going from right to left, ending at #
- Create a macro that examines a derivation encoding, checking that there are five sets of 1's with 0's in between. If this format is not there, return an X. If the start state is the same as the transition state, return a 1. If the start and transition states are not equal, return a 0. The single return character can have blanks after it until the next derivation.
- If we have returned an X, then erase all the input and return a 0.
- Else: transpose, move right, apply the derivation macro again. Continue until we have reached the end of the input.
- Apply the OR macro.

For deterministic TM's, the derivation macro would only need to return 0 or 1: on 0, the input is erased and 0 is returned; on 1, we can erase the 1 and keep applying the macro, and if we return a 1 from the rightmost derivation, we can return a 1 for the whole function.

12.
(a).
Ap. [Ab. [B(b) & ~S(p,p) ↔ S(b,p)]]
Where the domain is all people, S(x,y) is the predicate 'x shaves y', and B(x) is the predicate 'x is a barber'.
=
AbAp. (~(B(b) & ~S(p,p)) | S(b,p)) & (B(b) & ~S(p,p)) | ~S(b,p))
=
AbAp. (~B(b) | S(p,p) | S(b,p)) & (B(b) | ~S(b,p)) & (~S(p,p) | ~S(b,p))
=
{{~B(b), S(p,p), S(b,p)}, {~S(b,p), B(b)}, {~S(p,p), ~S(b,p))}
=
{{~B(b), S(b,p)}, {~S(b,p), B(b)}, {~S(b,p))} (resolved S(p,p))
=
{{~B(b)}, {B(b)}, {~S(b,p))} (resolved S(b,p))
=
{{~S(b,p))} (resolved B(b))
Should the conclusion be ~B(b)? If we conclude that nobody shaves anybody, then there must be no barbers. I'll probably play with this more later.

(b).
L(x,y) is the predicate 'x likes y'.
Ap. ~L(p,p) -> Ej. L(j,p)
=
Ap. L(p,p) | Ej. L(j,p)
=
Ap. L(p,p) | L(j(p),p) (skolemization)
=
{{L(p,p),L(j(p),p)}}

13.
This doesn't use the exact same syntactic rules from our book,
i.
Ex. Dragon(x)
ii.
Ex. Dragon(x) & (Sleeping(x) | Hunting(x))
iii.
Ex. Dragon(x) & (Hungry(x) -> ~Sleeping(x))
iv.
Ex. Dragon(x) & (Tired(x) -> ~Hunting(x))
(a).
Hunts in the forest (rules iii and ii)
(b).
Sleeping (rules iv and ii)

14.

```
slength([], 0).
slength([_|Xs], s(N)) :- slength(Xs, N).

?- length([a,b,[b,c]],X).
X = 3.
```

15.

```
flttn([],[]).
flttn([X|F], [X|Xs]) :- atomic(X), flttn(F, Xs).
flttn(F3, [X|Xs]) :- flttn(F1, X), flttn(F2, Xs), append(F1,F2,F3).
```

16.

(a). In these two situations, the cut doesn't have any particular advantage, and it will behave in the same way as the base case fact without a cut.

(b). In this case, the program would match X to [] and would not continue to compute further results, which is undesired if we want to compute all the possible pairs that could be appended to make [x,y,z]

17.

+ isn't in the environment but I'll assume it is implicitly
This will be bottom up, left to right type inference.
The formatting is pretty awful but hopefully the idea is all there.

```
A = {<1,int>, <zero,(int->bool)>, <pred,(int->int)>, <(+),(int->int->int)>}
A |- pred:(int->int)      A |- n:x [ide]
----------------------------------------
A |- pred(n):int [comb]             A.pred:(int->int) |- pred(n):(int->int) [ide]
----------------------------------------------------------------------------------------------------
A |- pred(pred(n)):int [comb]     A |- fib:(a->b) [ide] ((is this allowed?))
--------------------------------------------------------------------------------------------------
A |- fib(pred(pred(n))):b  [ide]   A |- pred(n):int [ide]    A |- fib:(a->b))
                                   --------------------------------------------------------------
                                   A |- fib(pred(n)):b [comb]   A |- +:int->int->int
-------------------------------------------------------------------------------------------------------
fib(pred(n))+fib(pred(pred(n)):int
[comb] [pretending that comb is defined for two param functions]
                                                     A |- pred(n):int     A |- zero:int-bool
                                                     ---------------------------------------------
A |- 1:int                                           A |- zero(pred(n)):bool  [comb]
----------------------------------------------------------------------------------------------------
A |- if zero(pred(n)) then 1 else fib(pred(n)) + fib(pred(pred(n))) : int [cond]

                                                     A |- zero:(int->bool)      A|-n:int
                                                     ---------------------------------------
                                                     A |- zero(n):bool
```

A |- 1:int

A |- if zero(n) then 4 else if zero(pred(n)) then 1 else fib(pred(n)+fib(pred(pred(n)))
 : int [cond] [let this conditional expression be called 'c']

A.n:int |- c:int

A |- (fun(n) c) : int -> int [abs]

A |- fib : int -> int [bind]

A.B |- fib :: B

A |- (rec fib) :: B

A |- rec fib:: B A.B |- (fun(n) c) : int

A |- (let rec fib in (fun(n) c): int

I was probably supposed to apply gen/spec in there somewhere, but I can't tell where.

18.
I have a few different ideas for projects that won't require all that much text to explain. Hopefully over the break I can get some feedback on them, narrow it down to one, and flesh out some specifics.

(1)
Creating an LALR(1) parser generator in Haskell or Smalltalk. I would probably rather do it in Haskell so that I can mess around with applicative and/or monadic parsing while also using the LALR(1) table. I had already begun to do this in Java for Clite -- I learned the concepts behind LR(1) parser table generation and the LR(1) parsing algorithm, but only implemented the very beginning of the project (grammar and production classes, LR(1) item classes). I was interested in possibly doing this in Smalltalk because the language looks really neat, but I could just do some smaller exercises on the side.

(2)
Designing a purely functional language that can appear object oriented and imperative. By default, the programmer would be inside the monad so could easily make an imperative looking function, but could also just as easily make a pure function. I think there could also be an interesting merging of Haskell-style data structures and objects, where the programmer defines type classes, and then object classes are algebraic data types which have instance implementations of the type classes, which could also be considered methods. With partial application, these methods/functions could then look equally like Haskell functions

and Smalltalk messages.

I have also played around with some unusual syntax for Clite that I would enjoy implementing. Functions have the form:

name : (param, param2) -> rettype = expression

or

name : (param, param2, etc) -> rettype {imperative/monadic; block; dec:int; dec := 4; }

(1) and (2) could perhaps be combined if it's not too much work, or maybe (1) as an SLR parser generator combined with (2).

(3)

I would be really, really excited about doing something using the distributed computing center, though I have much less of an idea about the specifics for this. I'm very interested in emergence and simulations, so perhaps something like a complex Game of Life, where genetic algorithms are inserted into an environment of some kind and must survive. Since I'm not totally sure what this might entail, since it's less related to the subjects in CNC, and since it's not very fleshed out, maybe this would be better as a project for next year.