1. Modified Horn Clause SLN Resolution Deduction System:

```
#lang racket
(require (lib "trace.ss"))

(define var? symbol?)

(define (apply-subst x e)
  (cond ((var? x) (let ((p (assoc x e))) (if p (cdr p) x)))
        ((pair? x) (cons (car x)(map (λ(y) (apply-subst y e)) (cdr
x))))
        (else x) ))

(define (unify x y e)
  (define (extend-subst v x e)
    (cons (cons v x)
          (map (λ(p) (cons (car p) (apply-subst (cdr p) (list (cons v
x))))) e)))
  (define (occur v x)
    (or (and (var? x) (eq? v x))
        (and (pair? x)(ormap (λ(y) (occur v y)) (cdr x))) ))
  (and e (let ((x (apply-subst x e))
               (y (apply-subst y e)))
           (cond ((equal? x y) e)
                 ((var? x) (and (not (occur x y)) (extend-subst x y
e)))
                 ((var? y) (and (not (occur y x)) (extend-subst y x
e)))
                 (else (and (pair? x) (pair? y) (=(length x) (length
y))
                            (equal? (car x) (car y)) ; test if
operator same
                            (foldl unify e (cdr x) (cdr y)) )) ))))

(define (freevarsin x)
  (cond ((var? x) (list x))
        ((pair? x) (remove-duplicates (append-map freevarsin (cdr
x))))
        (else '()) ))

(define (make-var-generator v)
  (define n 0)
  (λ()(set! n(+ 1 n))
    (string->symbol(string-append
                    (symbol->string v) (number->string(- n 1)))) ))

(define (rename ids body fresh-id)
  (apply-subst body(map(λ(id) (cons id (fresh-id))) ids)))
```

```
(define (resolve s c)
  (define (answer? c)
    (eq? 'answer (caadr c)))
  (define (move-answer-lit-to-front c)
    (append (filter (λ(l)(not(answer? l))) c) (filter answer? c)))
  (define (equate x y e) ; returns '() or ( e )
    (let ((t (unify x y e))) (if t(list t) '())))
  (define (equate-lit x y t)
    (cond ((and (¬? x) (not (¬? y))) (equate(cadr x) y t))
          ((and (¬? y) (not (¬? x))) (equate x (cadr y) t))
          (else '()) ))
  ;(printf "node => ~a\n" s)
  (map (λ(th) (instantiate-clause (move-answer-lit-to-front (append
(cdr s) (cdr c))) th))
       (equate-lit (car s) (car c) '()) ))

(define (¬? x) (and (pair? x) (eq? (car x) '¬)))

(define (freevarsin-clause c)
  (remove-duplicates(append-map freevarsin c)))

(define (rename-clause c fresh-id)
  (cdr (rename (freevarsin-clause c) (cons 'or c) fresh-id)))

(define (instantiate-clause c e)
  (map (λ(p) (apply-subst p e)) c))

(define (ATP axioms ¬conjecs)
  (define (move-pos-lit-to-front c)
    (append (filter (λ(l)(not(¬? l))) c) (filter ¬? c)))
  (define +axioms (map move-pos-lit-to-front axioms))
  ;(define fresh-id gensym) ;(make-var-generator 'x))
  (define fresh-id (make-var-generator 'x))
  (define (res-moves s)
    (append-map(λ(c) (map(λ(r) (list r c 1)) ; (s a w)
                         (resolve s (rename-clause c fresh-id))))
               +axioms))
  ;(define res-goal? null?)
  (define (goal? x) ; check if null or is just (¬(answer ...))
  (cond ((eq? x '()) #t)
        ((and (eq? 1 (length x)) (pair? x)) (eq? 'answer (first
(car(cdr(car x))))))
        (else #f)) )
  (define res-heuristic length)
  (A*-graph-search ¬conjecs goal? res-moves res-heuristic))

(require data/heap)
```

```
(define(A*-graph-search start goal? moves heuristic)
  (struct node (state pred move g h f))
  (define (node<=? x y) (<= (node-f x) (node-f y)))
  (define Q (make-heap node<=?))
  (define ht (make-hash))
  (define (not-in-hash? SAW) (eq? #f (hash-ref ht (get-hash (car
SAW)) #f)))
  (define (print-solution anode)
    (cond
      [(null? anode)]
      [(null? (node-pred anode)) (list (list 'start (node-state
anode) (node-g anode) (node-h anode) (node-f anode)))]
      [else (append (append (print-solution (node-pred anode))) (list
(list (node-move anode) (node-state anode) (node-g anode) (node-h
anode) (node-f anode))))]))
  (define (add-SAW-to-heap SAW prev)
    (let* [(weight (car (cdr (cdr SAW)))) (h (heuristic (car SAW)))
(action (car (cdr SAW)))]
      (let [(g (if (null? prev) weight (+ weight (node-g prev))))]
        (define f (+ g h))
        (if (< g 45) (node (car SAW) prev action g h f) '()))))
  (define (get-hash state) state)
  (define no-solution 'failure)

  (map (λ(x) (let [(h (heuristic x))] (heap-add! Q (node x '() 'start
0 h h)))) start)
  (let loop ()
    (define curr (heap-min Q))
    (hash-set! ht (get-hash (node-state curr)) curr)
    (heap-remove-min! Q)N
    ;(printf "~a\n" (length (node-state curr)))
    (cond
      [(> (heap-count Q) 150000) (printf "Over 100000 nodes in
heap\n") 'failure]
      [(goal? (node-state curr)) (print-solution curr)]
      [else
       (let ([SAWs (filter not-in-hash? (moves (node-state curr)))])
         (heap-add-all! Q (filter (λ(s) (not (null? s))) (map (λ(x)
(add-SAW-to-heap x curr)) SAWs))))
       (if (= (heap-count Q) 0) no-solution (loop))])))

(define block_axioms '(
                       ((poss(move b x y s)) (¬(on b x s)) (¬(clear b
s)) (¬(clear y s))
                                             (¬(block b)) (¬(block
y)) (¬(≠ b x)) (¬(≠ b y)) (¬(≠ x y)) )
                       ((on b y (move b x y s)) (¬(poss(move b x y
s))) )
```

```
                        ((clear x (move b x y s)) (¬(poss(move b x y
s))) )
                        ((on b2 x2(move b x y s)) (¬(on b2 x2 s)) (¬(≠
b2 b))
                                                  (¬(poss(move b x y
s))))
                        ((clear y2(move b x y s)) (¬(clear y2 s)) (¬(≠
y2 y))
                                                  (¬(poss(move b x y
s))))

                        ((poss(movetotable b x s)) (¬(on b x s))
(¬(clear b s))
                                                   (¬(block b))
(¬(block x)) (¬(≠ b x)))

                        ((on b(t) (movetotable b x s))
(¬(poss(movetotable b x s))))
                        ((clear x (movetotable b x s))
(¬(poss(movetotable b x s))))
                        ((on b2 x2(movetotable b x s)) (¬(on b2 x2 s))
(¬(≠ b2 b))

(¬(poss(movetotable b x s))) )
                        ((clear y2(movetotable b x s)) (¬(clear y2 s))
(¬(≠ y2 (t)))

(¬(poss(movetotable b x s))) )

                        ;;; distinct objects
                        ((≠(a)(b))) ((≠(b)(a)))
                        ((≠(a)(c))) ((≠(c)(a)))
                        ((≠(b)(c))) ((≠(c)(b)))
                        ((≠(a)(t))) ((≠(t)(a)))
                        ((≠(b)(t))) ((≠(t)(b)))
                        ((≠(c)(t))) ((≠(t)(c)))

                        ;;; blocks
                        ((block(a)))
                        ((block(b)))
                        ((block(c)))

                        ;;; fluents for initial state 0
                        ((on(b)(t)0))
                        ((on(a)(t)0))
                        ((on(c)(a)0))
                        ((clear(b)0))
                        ((clear(c)0))
```

```
                        ))

(define conj '(((¬(on(a)(b)s)) (¬(on(b)(c)s)) (¬(answer s)) )))
(define conj1 '(((¬(on(b)(c)s)) (¬(answer s)) )))
(define conj2 '(((¬(on(a)(b)s)) (¬(answer s)) )))

;(trace resolve)
(time (ATP block_axioms conj))
```

2. Output from Block Worlds example:

```
Welcome to DrRacket, version 5.2.1 [3m].
Language: racket; memory limit: 2048 MB.
cpu time: 30658 real time: 30709 gc time: 11752
'((start ((¬ (on (a) (b) s)) (¬ (answer s))) 0 2 2)
  (((on b y (move b x y s)) (¬ (poss (move b x y s)))) ((¬ (poss
(move (a) x6 (b) x7))) (¬ (answer (move (a) x6 (b) x7)))) 1 2 3)
  (((poss (move b x y s)) (¬ (on b x s)) (¬ (clear b s)) (¬ (clear y
s)) (¬ (block b)) (¬ (block y)) (¬ (≠ b x)) (¬ (≠ b y)) (¬ (≠ x y)))
   ((¬ (on (a) x6 x7)) (¬ (clear (a) x7)) (¬ (clear (b) x7)) (¬
(block (a))) (¬ (block (b))) (¬ (≠ (a) x6)) (¬ (≠ (a) (b))) (¬ (≠ x6
(b))) (¬ (answer (move (a) x6 (b) x7))))
   2
   9
   11)
  (((on b2 x2 (movetotable b x s)) (¬ (on b2 x2 s)) (¬ (≠ b2 b)) (¬
(poss (movetotable b x s))))
   ((¬ (on (a) x6 x651))
    (¬ (≠ (a) x649))
    (¬ (poss (movetotable x649 x650 x651)))
    (¬ (clear (a) (movetotable x649 x650 x651)))
    (¬ (clear (b) (movetotable x649 x650 x651)))
    (¬ (block (a)))
    (¬ (block (b)))
    (¬ (≠ (a) x6))
    (¬ (≠ (a) (b)))
    (¬ (≠ x6 (b)))
    (¬ (answer (move (a) x6 (b) (movetotable x649 x650 x651)))))
   3
   11
   14)
  (((on (a) (t) 0))
   ((¬ (≠ (a) x649))
    (¬ (poss (movetotable x649 x650 0)))
    (¬ (clear (a) (movetotable x649 x650 0)))
    (¬ (clear (b) (movetotable x649 x650 0)))
    (¬ (block (a)))
```

```
    (¬ (block (b)))
    (¬ (≠ (a) (t)))
    (¬ (≠ (a) (b)))
    (¬ (≠ (t) (b)))
    (¬ (answer (move (a) (t) (b) (movetotable x649 x650 0)))))
   4
   10
   14)
  (((≠ (a) (c)))
   ((¬ (poss (movetotable (c) x650 0)))
    (¬ (clear (a) (movetotable (c) x650 0)))
    (¬ (clear (b) (movetotable (c) x650 0)))
    (¬ (block (a)))
    (¬ (block (b)))
    (¬ (≠ (a) (t)))
    (¬ (≠ (a) (b)))
    (¬ (≠ (t) (b)))
    (¬ (answer (move (a) (t) (b) (movetotable (c) x650 0)))))
   5
   9
   14)
  (((poss (movetotable b x s)) (¬ (on b x s)) (¬ (clear b s)) (¬
(block b)) (¬ (block x)) (¬ (≠ b x)))
   ((¬ (on (c) x650 0))
    (¬ (clear (c) 0))
    (¬ (block (c)))
    (¬ (block x650))
    (¬ (≠ (c) x650))
    (¬ (clear (a) (movetotable (c) x650 0)))
    (¬ (clear (b) (movetotable (c) x650 0)))
    (¬ (block (a)))
    (¬ (block (b)))
    (¬ (≠ (a) (t)))
    (¬ (≠ (a) (b)))
    (¬ (≠ (t) (b)))
    (¬ (answer (move (a) (t) (b) (movetotable (c) x650 0)))))
   6
   13
   19)
  (((on (c) (a) 0))
   ((¬ (clear (c) 0))
    (¬ (block (c)))
    (¬ (block (a)))
    (¬ (≠ (c) (a)))
    (¬ (clear (a) (movetotable (c) (a) 0)))
    (¬ (clear (b) (movetotable (c) (a) 0)))
    (¬ (block (b)))
    (¬ (≠ (a) (t)))
```

```
  (¬ (≠ (a) (b)))
  (¬ (≠ (t) (b)))
  (¬ (answer (move (a) (t) (b) (movetotable (c) (a) 0)))))
 7
 11
 18)
(((clear (c) 0))
 ((¬ (block (c)))
  (¬ (block (a)))
  (¬ (≠ (c) (a)))
  (¬ (clear (a) (movetotable (c) (a) 0)))
  (¬ (clear (b) (movetotable (c) (a) 0)))
  (¬ (block (b)))
  (¬ (≠ (a) (t)))
  (¬ (≠ (a) (b)))
  (¬ (≠ (t) (b)))
  (¬ (answer (move (a) (t) (b) (movetotable (c) (a) 0)))))
 8
 10
 18)
(((block (c)))
 ((¬ (block (a)))
  (¬ (≠ (c) (a)))
  (¬ (clear (a) (movetotable (c) (a) 0)))
  (¬ (clear (b) (movetotable (c) (a) 0)))
  (¬ (block (b)))
  (¬ (≠ (a) (t)))
  (¬ (≠ (a) (b)))
  (¬ (≠ (t) (b)))
  (¬ (answer (move (a) (t) (b) (movetotable (c) (a) 0)))))
 9
 9
 18)
(((block (a)))
 ((¬ (≠ (c) (a)))
  (¬ (clear (a) (movetotable (c) (a) 0)))
  (¬ (clear (b) (movetotable (c) (a) 0)))
  (¬ (block (b)))
  (¬ (≠ (a) (t)))
  (¬ (≠ (a) (b)))
  (¬ (≠ (t) (b)))
  (¬ (answer (move (a) (t) (b) (movetotable (c) (a) 0)))))
 10
 8
 18)
(((≠ (c) (a)))
 ((¬ (clear (a) (movetotable (c) (a) 0)))
  (¬ (clear (b) (movetotable (c) (a) 0)))
```

```
    (¬ (block (b)))
    (¬ (≠ (a) (t)))
    (¬ (≠ (a) (b)))
    (¬ (≠ (t) (b)))
    (¬ (answer (move (a) (t) (b) (movetotable (c) (a) 0)))))
   11
   7
   18)
  (((clear x (movetotable b x s)) (¬ (poss (movetotable b x s))))
   ((¬ (poss (movetotable (c) (a) 0)))
    (¬ (clear (b) (movetotable (c) (a) 0)))
    (¬ (block (b)))
    (¬ (≠ (a) (t)))
    (¬ (≠ (a) (b)))
    (¬ (≠ (t) (b)))
    (¬ (answer (move (a) (t) (b) (movetotable (c) (a) 0)))))
   12
   7
   19)
  (((poss (movetotable b x s)) (¬ (on b x s)) (¬ (clear b s)) (¬
(block b)) (¬ (block x)) (¬ (≠ b x)))
   ((¬ (on (c) (a) 0))
    (¬ (clear (c) 0))
    (¬ (block (c)))
    (¬ (block (a)))
    (¬ (≠ (c) (a)))
    (¬ (clear (b) (movetotable (c) (a) 0)))
    (¬ (block (b)))
    (¬ (≠ (a) (t)))
    (¬ (≠ (a) (b)))
    (¬ (≠ (t) (b)))
    (¬ (answer (move (a) (t) (b) (movetotable (c) (a) 0)))))
   13
   11
   24)
  (((on (c) (a) 0))
   ((¬ (clear (c) 0))
    (¬ (block (c)))
    (¬ (block (a)))
    (¬ (≠ (c) (a)))
    (¬ (clear (b) (movetotable (c) (a) 0)))
    (¬ (block (b)))
    (¬ (≠ (a) (t)))
    (¬ (≠ (a) (b)))
    (¬ (≠ (t) (b)))
    (¬ (answer (move (a) (t) (b) (movetotable (c) (a) 0)))))
   14
   10
```

```
   24)
  (((clear (c) 0))
   ((¬ (block (c)))
    (¬ (block (a)))
    (¬ (≠ (c) (a)))
    (¬ (clear (b) (movetotable (c) (a) 0)))
    (¬ (block (b)))
    (¬ (≠ (a) (t)))
    (¬ (≠ (a) (b)))
    (¬ (≠ (t) (b)))
    (¬ (answer (move (a) (t) (b) (movetotable (c) (a) 0)))))
   15
   9
   24)
  (((block (c)))
   ((¬ (block (a))) (¬ (≠ (c) (a))) (¬ (clear (b) (movetotable (c)
(a) 0))) (¬ (block (b))) (¬ (≠ (a) (t))) (¬ (≠ (a) (b))) (¬ (≠ (t)
(b))) (¬ (answer (move (a) (t) (b) (movetotable (c) (a) 0)))))
   16
   8
   24)
  (((block (a)))
   ((¬ (≠ (c) (a))) (¬ (clear (b) (movetotable (c) (a) 0))) (¬ (block
(b))) (¬ (≠ (a) (t))) (¬ (≠ (a) (b))) (¬ (≠ (t) (b))) (¬ (answer
(move (a) (t) (b) (movetotable (c) (a) 0)))))
   17
   7
   24)
  (((≠ (c) (a))) ((¬ (clear (b) (movetotable (c) (a) 0))) (¬ (block
(b))) (¬ (≠ (a) (t))) (¬ (≠ (a) (b))) (¬ (≠ (t) (b))) (¬ (answer
(move (a) (t) (b) (movetotable (c) (a) 0))))) 18 6 24)
  (((clear y2 (movetotable b x s)) (¬ (clear y2 s)) (¬ (≠ y2 (t))) (¬
(poss (movetotable b x s))))
   ((¬ (clear (b) 0)) (¬ (≠ (b) (t))) (¬ (poss (movetotable (c) (a)
0))) (¬ (block (b))) (¬ (≠ (a) (t))) (¬ (≠ (a) (b))) (¬ (≠ (t) (b)))
(¬ (answer (move (a) (t) (b) (movetotable (c) (a) 0)))))
   19
   8
   27)
  (((clear (b) 0))
   ((¬ (≠ (b) (t))) (¬ (poss (movetotable (c) (a) 0))) (¬ (block
(b))) (¬ (≠ (a) (t))) (¬ (≠ (a) (b))) (¬ (≠ (t) (b))) (¬ (answer
(move (a) (t) (b) (movetotable (c) (a) 0)))))
   20
   7
   27)
  (((≠ (b) (t))) ((¬ (poss (movetotable (c) (a) 0))) (¬ (block (b)))
(¬ (≠ (a) (t))) (¬ (≠ (a) (b))) (¬ (≠ (t) (b))) (¬ (answer (move (a)
```

```
(t) (b) (movetotable (c) (a) 0))))) 21 6 27)
  (((poss (movetotable b x s)) (¬ (on b x s)) (¬ (clear b s)) (¬
(block b)) (¬ (block x)) (¬ (≠ b x)))
   ((¬ (on (c) (a) 0))
    (¬ (clear (c) 0))
    (¬ (block (c)))
    (¬ (block (a)))
    (¬ (≠ (c) (a)))
    (¬ (block (b)))
    (¬ (≠ (a) (t)))
    (¬ (≠ (a) (b)))
    (¬ (≠ (t) (b)))
    (¬ (answer (move (a) (t) (b) (movetotable (c) (a) 0)))))
   22
   10
   32)
  (((on (c) (a) 0))
   ((¬ (clear (c) 0)) (¬ (block (c))) (¬ (block (a))) (¬ (≠ (c) (a)))
(¬ (block (b))) (¬ (≠ (a) (t))) (¬ (≠ (a) (b))) (¬ (≠ (t) (b))) (¬
(answer (move (a) (t) (b) (movetotable (c) (a) 0)))))
   23
   9
   32)
  (((clear (c) 0))
   ((¬ (block (c))) (¬ (block (a))) (¬ (≠ (c) (a))) (¬ (block (b)))
(¬ (≠ (a) (t))) (¬ (≠ (a) (b))) (¬ (≠ (t) (b))) (¬ (answer (move (a)
(t) (b) (movetotable (c) (a) 0)))))
   24
   8
   32)
  (((block (c))) ((¬ (block (a))) (¬ (≠ (c) (a))) (¬ (block (b))) (¬
(≠ (a) (t))) (¬ (≠ (a) (b))) (¬ (≠ (t) (b))) (¬ (answer (move (a) (t)
(b) (movetotable (c) (a) 0))))) 25 7 32)
  (((block (a))) ((¬ (≠ (c) (a))) (¬ (block (b))) (¬ (≠ (a) (t))) (¬
(≠ (a) (b))) (¬ (≠ (t) (b))) (¬ (answer (move (a) (t) (b)
(movetotable (c) (a) 0)))))) 26 6 32)
  (((≠ (c) (a))) ((¬ (block (b))) (¬ (≠ (a) (t))) (¬ (≠ (a) (b))) (¬
(≠ (t) (b))) (¬ (answer (move (a) (t) (b) (movetotable (c) (a) 0)))))
27 5 32)
  (((block (b))) ((¬ (≠ (a) (t))) (¬ (≠ (a) (b))) (¬ (≠ (t) (b))) (¬
(answer (move (a) (t) (b) (movetotable (c) (a) 0))))) 28 4 32)
  (((≠ (a) (t))) ((¬ (≠ (a) (b))) (¬ (≠ (t) (b))) (¬ (answer (move
(a) (t) (b) (movetotable (c) (a) 0)))))) 29 3 32)
  (((≠ (a) (b))) ((¬ (≠ (t) (b))) (¬ (answer (move (a) (t) (b)
(movetotable (c) (a) 0)))))) 30 2 32)
  (((≠ (t) (b))) ((¬ (answer (move (a) (t) (b) (movetotable (c) (a)
0)))))) 31 1 32))
```

```
cpu time: 12 real time: 11 gc time: 0
'((start ((¬ (on (b) (c) s)) (¬ (answer s))) 0 2 2)
  (((on b y (move b x y s)) (¬ (poss (move b x y s)))) ((¬ (poss
(move (b) x6 (c) x7))) (¬ (answer (move (b) x6 (c) x7)))) 1 2 3)
  (((poss (move b x y s)) (¬ (on b x s)) (¬ (clear b s)) (¬ (clear y
s)) (¬ (block b)) (¬ (block y)) (¬ (≠ b x)) (¬ (≠ b y)) (¬ (≠ x y)))
   ((¬ (on (b) x6 x7)) (¬ (clear (b) x7)) (¬ (clear (c) x7)) (¬
(block (b))) (¬ (block (c))) (¬ (≠ (b) x6)) (¬ (≠ (b) (c))) (¬ (≠ x6
(c))) (¬ (answer (move (b) x6 (c) x7))))
   2
   9
   11)
  (((on (b) (t) 0)) ((¬ (clear (b) 0)) (¬ (clear (c) 0)) (¬ (block
(b))) (¬ (block (c))) (¬ (≠ (b) (t))) (¬ (≠ (b) (c))) (¬ (≠ (t) (c)))
(¬ (answer (move (b) (t) (c) 0)))) 3 8 11)
  (((clear (b) 0)) ((¬ (clear (c) 0)) (¬ (block (b))) (¬ (block (c)))
(¬ (≠ (b) (t))) (¬ (≠ (b) (c))) (¬ (≠ (t) (c))) (¬ (answer (move (b)
(t) (c) 0)))) 4 7 11)
  (((clear (c) 0)) ((¬ (block (b))) (¬ (block (c))) (¬ (≠ (b) (t)))
(¬ (≠ (b) (c))) (¬ (≠ (t) (c))) (¬ (answer (move (b) (t) (c) 0)))) 5
6 11)
  (((block (b))) ((¬ (block (c))) (¬ (≠ (b) (t))) (¬ (≠ (b) (c))) (¬
(≠ (t) (c))) (¬ (answer (move (b) (t) (c) 0)))) 6 5 11)
  (((block (c))) ((¬ (≠ (b) (t))) (¬ (≠ (b) (c))) (¬ (≠ (t) (c))) (¬
(answer (move (b) (t) (c) 0)))) 7 4 11)
  (((≠ (b) (t))) ((¬ (≠ (b) (c))) (¬ (≠ (t) (c))) (¬ (answer (move
(b) (t) (c) 0)))) 8 3 11)
  (((≠ (b) (c))) ((¬ (≠ (t) (c))) (¬ (answer (move (b) (t) (c) 0))))
9 2 11)
  (((≠ (t) (c))) ((¬ (answer (move (b) (t) (c) 0)))) 10 1 11))
```

3. Axiomatize the Air Cargo Transport Example.

```
; Initial State
((At (C1, SFO)))
((At (C2, JFK)))
((At (P1, SFO)))
((At (P2, FJK)))
((Cargo C1))
((Cargo C2))
((Plane P1))
((Plane P2))
((Airport SFO))
((Airport JFK))

; Distinct Objects
((≠ P1 P2)) ((≠ P2 P1))
((≠ P1 SFO)) ((≠ SFO P1))
```

```
((≠ P1 JFK)) ((≠ JFK P1))
((≠ P1 C1)) ((≠ C1 P1))
((≠ P1 C2)) ((≠ C2 P1))
((≠ P2 SFO)) ((≠ SFO P2))
((≠ P2 JFK)) ((≠ JFK P2))
((≠ P2 C1)) ((≠ C1 P2))
((≠ P2 C2)) ((≠ C2 P2))
((≠ SFO JFK)) ((≠ JFK SFO))
((≠ SFO C1)) ((≠ C1 SFO))
((≠ SFO C2)) ((≠ C2 SFO))
((≠ JFK C1)) ((≠ C1 JFK))
((≠ JFK C2)) ((≠ C2 JFK))
((≠ C1 C2)) ((≠ C2 C1))

; Planes
((Plane P1)) ((Plane P2))

; Cargo
((Cargo C1)) ((Cargo C2))

; Airports
((Airport SFO)) ((Airport JFK))

; Load Action
((poss(load c p a)

((poss(Load c p a s)) (¬(At c a s)) (¬(At p a s)) (¬(Cargo c))
(¬(Plane p)) (¬(Airport a)))
((In c p (Load c p a s)) (¬(poss(Load c p a s))))      ; result
((At c2 a2 (Load c p a s)) (¬(At c2 a2 s)) (¬(≠ c2 c)) ; frame law
                           (¬(poss(Load c p a s))))

((poss(Unload c p a s)) (¬(In c p s)) (¬(At p a s)) (¬(Cargo c))
(¬(Plane p)) (¬(Airport a)))
((At c a (Unload c p a s)) (¬(poss(Unload c p a s))))    ; result
((In c2 p2 (Unload c p a s)) (¬(In c2 p2 s)) (¬(≠ c2 c)) ; frame law
                             (¬(poss(Unload c p a s))))

((poss(Fly p from to s)) (¬(At p from s)) (¬(Plane p)) (¬(Airport
from)) (¬(Airport to)))
((At p to (Fly p from to s)) (¬(poss(Fly p from to s))))       ;
result
((At p2 from2 (Fly p from to s)) (¬(At p2 from2 s)) (¬(≠ p2 p)) ;
frame law
                                 (¬(poss(Fly p from to s))))
```

Re-ordering can help, but it can also hurt the planning. The order of the negated literals

changes the order in which the planner walks the problem graph. Re-ordering the clauses in the set of clauses, changes slightly the path walked on the graph because the earlier clauses will be resolved first, but the heuristic will still choose the resolved clauses with the fewest number of literals. For the blocks problem a better heuristic would calculate how many blocks are out of position, or how many blocks are under or over the incorrect block. For the plane problem, a better heuristic might be counting how many planes and packages are at of place.

4. (a) What are the major problems with this Situational Calculus representation of planning?

Situational Calculus provides an expressive and powerful means to describe an environment and goals for a planner. This same power and expressiveness is also its Achilles heel because it is too general to allow the use of more powerful heuristic functions. By lifting the representation to a subset of first-order logic, some of the expressive is lost, but in return, planners can derive domain-independent heuristics. The trade-off of a restricted representation schema for better heuristics is a good one because many of the problems in planning require polynomial space. Employing an accurate heuristics cuts down on the number of states significantly and can make a problem, which was previously unsolvable with a modern computer, solvable. The second major weakness of the Situational Calculus representation is the frame laws, which are hard to write and are easily broken: if any information about the problem domain is changed, then significant effort will be required to update the frame laws, which are likely to change drastically.

(b) How would you write a STRIPS like planning program?

To implement a STRIPS-like planning program I would begin from the goal state and work backwards to the initial state. This allows the planner to focus on the steps that are relevant, meaning they could have been the last step in a plan leading to the current state, which is important because planning problems often require polynomial space. The states that are considered relevant are the states that unify with at least one element of the goal, and At each state, the planner would search for a grounded

action, one with all variables bound to literals, that satisfies a precondition of the goal state. The next (really the previous state, since working backwards) could then be determined from g' = (g – ADD(a)) ∪ Precond(a), where ADD(a) is the list of fluents that are positive literals in the action's effects. In the STRIPS planner it is easier to derive domain-independent heuristic functions. For example, to relax the constraints, the preconditions for each state can be ignored—every actions can be applied in every state. The heuristic would then record the minimum number of moves required with these relaxed constraints.