Question 1   7.20

Answer:

S1: (~A v B v E) ^ (~B v A) ^ (~E v A)

S2: (~E v D)

S3: (~C v ~F v ~B)

S4: (~E v B)

S5: (~B v F)

S6: (~B v C)

Question 2   7.12

Answer:

Let KB denote {Si, 1<=i<=6}. In order to prove (alpha = ~A ^ ~B), we can conduct the PL-Resolution algorithm to KB ^ ~alpha, namely adding the **negation of alpha:**

S7: (A v B)

and for convenience, break S1 into:

S01: (~A v B v E)

S02: (~B v A)

S03: (~E v A)

That gives us the set for resolution:

S01: (~A v B v E)

S02: (~B v A)

S03: (~E v A)

S2: (~E v D)

S3: (~C v ~F v ~B)

S4: (~E v B)

S5: (~B v F)

S6: (~B v C)

S7: (A v B)

Thus, PL-Resolution {S11, S12, S13, S2, S3, S4, S5, S6, S7} :

Resolve{ S7, S6 } -> S8: (A v C)
Resolve{ S7, S02 } -> S9: (A)

Resolve{ S4, S01 } -> S10 : (~A v B)
Resolve{ S9, S10 } -> S11: (B)
Resolve{ S11, S3 } -> S12: (~C v ~F)
Resolve{ S12, S5 } -> S13: (~B v ~C)
Resolve{ S13, S6 } -> S14: (~B)
Resolve{ S14, S11 } -> S15: ()   -> This is an empty set

Therefore, KB ^ (~alpha) is **unsatisfiable;** by contradiction, **KB entails alpha = (~A ^ ~B)**

Question 3

The program reads the test cases from the **assn3test.rkt** file, converts and do resolution. In the resolution part, it not only ensures the correctness of the results, but also contains quite a few improvements on its **efficiency**.

Code Section:

```
(include "./assn3test.rkt")

;----------- CNF-Convert------------;
; covert all symbol input into string
(define (all-to-string my-symbol)
  (cond [(null? my-symbol) ""]
        [else (if (list? (car my-symbol))
                  (string-append "(" (symbol->string (car (car my-symbol)))
                                 (all-to-string (cdr (car my-symbol))) ")")
                  (string-append (symbol->string (car my-symbol))
                                 (all-to-string (cdr my-symbol))))]))

; invert a clause
(define (invert P)
  (define result P)
  (set! result (regexp-replace* "\\^" result "!"))
  (set! result (regexp-replace* "\\/" result "^"))
  (set! result (regexp-replace* "!" result "/"))
  (set! result (regexp-replace* "[a-zA-Z0-9]+" result
                                (lambda (all)
                                  (string-append "~" all))))
  (set! result (regexp-replace* "~~" result ""))
  result
)

; when flag = 1, convert query, 0 convert KB
(define (CNF-Convert clauses flag)
  (define result '())
  (define queries '())
  (define Q '())
  (for ([clause clauses])
    (set! result (append result (list (CNF (all-to-string clause))))))

  ; formalize the queries as list of lists, and negate literals
  (if (equal? 1 flag)
      (begin
      (for ([i result])
        (when (not (null? Q))
          (set! queries (append queries (list Q))))
```

```
        (set! Q '())
        (for ([j i])
          ;(displayln j)
          (set! Q
          (append Q
          (list (list(string->symbol
             ((lambda (l)
             (if (equal? (string-ref l 0) #\~)
               (string-trim l "~")
               (string-append "~" l)))
               (symbol->string j)))))))))
            (append queries (list Q)))
  result))

; convert on top of strings
(define (CNF sentence)
  (define P "")
  (define Q "")
  (cond [(regexp-match? #rx"<=>" sentence)        ; lowest precedence "<=>"
          (begin
            (set! P (list-ref(regexp-split #rx"<=>" sentence) 0))
            (set! Q (list-ref(regexp-split #rx"<=>" sentence) 1))
            (list (CNF (string-append P "=>" Q)) (CNF (string-append Q "=>" P))))]
        [(regexp-match? #rx"=>" sentence)         ; second lowest precedence "<=>"
          (begin
            (set! P (list-ref(regexp-split #rx"=>" sentence) 0))
            (set! Q (list-ref(regexp-split #rx"=>" sentence) 1))
            (append (CNF (invert P)) (CNF Q)))]
        [(regexp-match? #rx"~(\\(.+\\))" sentence) ; the parenthesis with negation
          (CNF (regexp-replace* "[\\(\\)]" (invert (list-ref (regexp-match #rx"~(\\(.+\\))" sentence) 1)) ""))]
                                              ; need to remove the "(" and ")" after invert
        [(regexp-match? #rx"(\\(.+\\))" sentence)  ; the parenthesis
          (CNF (regexp-replace* "[\\(\\)]" (list-ref (regexp-match #rx"~(\\(.+\\))" sentence) 1) ""))]
        [(regexp-match? #rx"\\/" sentence)        ; the union "/"
          (begin
            (set! P (list-ref(regexp-split #rx"\\/" sentence) 0))
            (set! Q (list-ref(regexp-split #rx"\\/" sentence) 1))
            (list (string->symbol P) (string->symbol Q)))]
        [(regexp-match? #rx"\\^" sentence)        ; the intersection "^"
          (begin
            (set! P (list-ref(regexp-split #rx"\\^" sentence) 0))
            (set! Q (list-ref(regexp-split #rx"\\^" sentence) 1))
            (list (string->symbol P) (string->symbol Q)))]
        [(regexp-match? #rx"~~" sentence)         ; remove double negations
            (list(string->symbol (regexp-replace* "~~" sentence "")))  ]
        [else (list (string->symbol sentence))]
        ))

;----------- PL-Resolution ------------;

(define (complementary-literals? l1 l2)
  (let* ([l1str (symbol->string l1)]
         [l2str (symbol->string l2)]
         [l1neg (equal? #\~ (string-ref l1str 0))]
         [l2neg (equal? #\~ (string-ref l2str 0))]
         [symbol1 (if l1neg (substring l1str 1) l1str)]
         [symbol2 (if l2neg (substring l2str 1) l2str)])
    (and (xor l1neg l2neg)
```

```
        (equal? symbol1 symbol2))))

(define (contain-empty? clauses)          ; check if contains empty clause
  (define flag #f)
  (for/or ([i clauses])
    (when (null? i) (set! flag #t)) flag)
  flag)

(define (remove-exist clauses new_clause)  ; remove those clauses already exist
  (define result new_clause)
  (if (equal? new_clause '(#f)) '()
  (begin
    (for ([i clauses])
      (for ([j new_clause])
        (when (equal? (list->set i) (list->set j)) (set! result (remove j result))))) result)))
                                        ; **use hash set to avoid duplication

(define (PL-Resolve C1 C2)
  (define resolvent '(#f))               ; the resolvent is initialized #f, to represent the case as no resolvent
  (for ([i C1])
    (for ([j C2])
      (cond [(complementary-literals? i j) ; once find a complementary pair, add to the resolvents
             (if (equal? resolvent '(#f)) (set! resolvent (list (remove-duplicates (append (remove i C1) (remove j C2)))))
                          (set! resolvent (append resolvent (list (remove-duplicates (append (remove i C1) (remove j C2)))))))]
            [else #t])))
  resolvent)

(define (PL-Resolution KB alpha num)
  (define clauses (append alpha KB))      ; **Left-append alpha
  (if (null? KB) (if (loop alpha)         ; handle the case when KB is empty
                  (fprintf (current-output-port)
                           "KB entails query ~s\n" num)
                  (fprintf (current-output-port)
                           "KB does not entail query ~s\n" num))
  (if (loop clauses) (fprintf (current-output-port)  ; the recursive call
                           "KB entails query ~s\n" num)
                  (fprintf (current-output-port)
                           "KB does not entail query ~s\n" num))))

(define (loop clauses)
  (define L (length clauses))
  (define resolvent '(#f))
  ;(displayln clauses)                    ; uncomment this to show the clauses each iterations
  (cond [(contain-empty? clauses) #t]
        [else (begin
                (for ([i (in-range L)])
                  (for ([j (in-range (add1 i) L)])
                    #:break (not (equal? '(#f) resolvent))            ; **keep a focus on the goal
                    (set! resolvent (remove-exist clauses (PL-Resolve (list-ref clauses i) (list-ref clauses j))))
                    (cond [(null? resolvent) (set! resolvent '(#f))]   ; if all new clauses already exist, ignore
                          [else (when (not (equal? resolvent '(#f)))   ; and reset resolvent
                                  (set! clauses (append resolvent clauses)))]))) ; **left append the new resolvent
                (if (equal? resolvent '(#f)) #f (loop (sort clauses (lambda (x y) (< (length x) (length y)))))))]))
                                                  ; **sort the clauses according to the length of clause
                                                  ; to propogate the unit or short clauses
```

As shown in the code and code comments (marked as **), there are several implementations to **improve the efficiency** of PL-Resolution, which includes:

1. **Eliminate redundant clauses** by not adding duplicate resolvent to the clauses. Use list->set transform the two clauses before comparing them, so that more possible redundant are avoided.

2. **Left-append alpha**, namely start the resolution with the query.

3. **Keep a focus on the goal**, namely every time we find new resolvents, we left-append them and start resolution from them.

4. **Propaget the unit clauses** as much as possible by sorting the clauses according to the length of clause .

And as we will see in the test session, these methods indeed improve the efficiency of resolution.

Testing Section:

Firstly, the supplied tests:

```
(PL-Resolution (CNF-Convert KB 0) (list-ref (CNF-Convert queries 1) 0) 0)
(PL-Resolution (CNF-Convert KB 0) (list-ref (CNF-Convert queries 1) 1) 1)
(PL-Resolution (CNF-Convert KB 0) (list-ref (CNF-Convert queries 1) 2) 2)
(PL-Resolution (CNF-Convert KB 0) (list-ref (CNF-Convert queries 1) 3) 3)
(PL-Resolution (CNF-Convert KB 0) (list-ref (CNF-Convert queries 1) 4) 4)
```

Output:

KB entails query 0
KB does not entail query 1
KB does not entail query 2
KB does not entail query 3
KB entails query 4

We can display the clauses each iteration to verify its correctness as well as see how it improves efficiency:

((~Girl) (FirstGrade) (~FirstGrade Child) (~Child ~Male Boy) (~Kindergarten Child) (~Child ~Female Girl) (Female))
((~Girl) (FirstGrade) (Female) (~Child ~Female) (~FirstGrade Child) (~Kindergarten Child) (~Child ~Male Boy) (~Child ~Female Girl))
((Child) (~Girl) (FirstGrade) (Female) (~Child ~Female) (~FirstGrade Child) (~Kindergarten Child) (~Child ~Male Boy) (~Child ~Female Girl))
((~Female) (Child) (~Girl) (FirstGrade) (Female) (~Child ~Female) (~FirstGrade Child) (~Kindergarten Child) (~Child ~Male Boy) (~Child ~Female Girl))
(() (~Female) (Child) (~Girl) (FirstGrade) (Female) (~Child ~Female) (~FirstGrade Child) (~Kindergarten Child) (~Child ~Male Boy) (~Child ~Female Girl))

As it shows, it only takes 5 iterations/resolutions to obtain the empty clause. The query was left-appended to the KB, all the clauses are sorted by their length, and there are no duplications.

Secondly, the corner tests:

Define some KBs and queries:

```
(displayln "\nOther corner cases:\n")
(define KB_CNF_test_1
  '((FirstGrade) (~FirstGrade)))   ;  contradictory KB

(define KB_CNF_test_2
  '((FirstGrade ~FirstGrade)))      ;  tautology KB
```

```
(define KB_CNF_test_3
  '())                                   ;  empty KB

(define KB_CNF_test_4            ; to test with fifth query
  '((q p)))

(define queries_N_CNF_test  ;queries are presented as ~query in CNF,
  '(((FirstGrade) (~Boy))            ; contingencies query
    ((FirstGrade) (~FirstGrade))    ; tautology query
    ((~Boy Boy))                          ; contradictory query
    ((FirstGrade))                        ; single literal query
    ((~q ~p))))                           ; to test with KB_4
```

Do the test:

Given a contradictory KB

```
(PL-Resolution KB_CNF_test_1 (list-ref queries_N_CNF_test 0) 5)   ; given a contradictory KB, contingencies query
(PL-Resolution KB_CNF_test_1 (list-ref queries_N_CNF_test 1) 6)   ; given a contradictory KB, tautology query
(PL-Resolution KB_CNF_test_1 (list-ref queries_N_CNF_test 2) 7)   ; given a contradictory KB, contradictory query
```

Output:

KB entails query 5
KB entails query 6
KB entails query 7

Given a tautology KB

```
(PL-Resolution KB_CNF_test_2 (list-ref queries_N_CNF_test 0) 8)    ; given a tautology KB, contingencies query
(PL-Resolution KB_CNF_test_2 (list-ref queries_N_CNF_test 1) 9)    ; given a tautology KB, tautology query
(PL-Resolution KB_CNF_test_2 (list-ref queries_N_CNF_test 2) 10)  ; given a tautology KB, contradictory query
```

Output:

KB does not entail query 8
KB entails query 9
KB does not entail query 10

Given an empty KB

```
(PL-Resolution KB_CNF_test_3 (list-ref queries_N_CNF_test 0) 11)  ; given an empty KB, contingencies query
(PL-Resolution KB_CNF_test_3 (list-ref queries_N_CNF_test 1) 12)  ; given an empty KB, tautology query
(PL-Resolution KB_CNF_test_3 (list-ref queries_N_CNF_test 2) 13)  ; given an empty KB, contradictory query
```

Output:

KB does not entail query 11
KB entails query 12
KB does not entail query 13

Given a contingencies KB

```
(PL-Resolution KB_CNF (list-ref queries_N_CNF_test 0) 14)     ; given a contingencies KB, contingencies query
(PL-Resolution KB_CNF (list-ref queries_N_CNF_test 1) 15)     ; given a contingencies KB, tautology query
(PL-Resolution KB_CNF (list-ref queries_N_CNF_test 2) 16)     ; given a contingencies KB, contradictory query
```

Output:

KB does not entail query 14
KB entails query 15
KB does not entail query 16

A case with multiple complementary pairs '((q p)) and ((~q ~p))

(PL-Resolution KB_CNF_test_4 (list-ref queries_N_CNF_test 4) 17)   ; the case with multiple complementary pairs

Output:

KB does not entail query 17

Finally, we use the Question 2 for testing:

```
; the case in Question 2
(define clauses
 '((~A B E)
  (~B A)
  (~E A)
  (~E D)
  (~C ~F ~B)
  (~E B)
  (~B F)
  (~B C)
  (A B))
  )

(define new_clause
  '((A B))
  )

(PL-Resolution clauses new_clause 18)
```

Output:

KB entails query 18

Test for converter: (Only outputs here)

Test for converter, show the correctness of the CNF conversion:

KB:
original: ((FirstGrade) (FirstGrade => Child) (Child ^ Male => Boy) (Kindergarten => Child) (Child ^ Female => Girl) (Female))
in CNF: ((FirstGrade) (~FirstGrade Child) (~Child ~Male Boy) (~Kindergarten Child) (~Child ~Female Girl) (Female))

Queries:
original: ((Girl) (~ Boy) (~~Boy) (~ (FirstGrade ^ ~ ~ Girl)) (Boy / Child))
in CNF: (((~Girl)) ((Boy)) ((~Boy)) ((FirstGrade) (Girl)) ((~Boy) (~Child)))

Some other tests:
original: ((A=>B^C) (~ (A/B)) (A^C<=>~B^D))
in CNF: ((~A B C) (~A ~B) ((~A ~C ~B D) (B ~D A C)))