Helper Functions:

```
(defvar *predicate-list* nil)
(defvar *query-list* nil)
(defvar *fact-list* nil)
(defvar *line-list* nil)
;;reads file and stores all content in a single string
(defun getFile(filename)
    (let ((in (open filename :if-does-not-exist nil))(str "")(predicate-flag 0)(fact-flag 0)(query-flag 0))
        (when in
            (loop for line = (read-line in nil)
                while line do (setq *line-list* (append *line-list* (list line)))
            )
            (close in)
        )
        str
    )
)
;;gets rid of unnecessary spaces
(defun clearList ()
  (let ((tempString "")(temp-list nil)(first-flag 0) (last-flag 0) (space-count 0))
    (loop for x in *line-list*
      do(loop for c across x
        do(if (eq c #\Space) (setq space-count (+ space-count 1)) (setq space-count 0))
        do(if (< space-count 2) (setq tempString (concatenate 'string tempString (string c))))
      )
      do(setq temp-list (append temp-list (list tempString)))
      do(setq tempString "")
    )


    temp-list
  )
)
```

getFile: Reads file line by line and appends each line to global line-list.

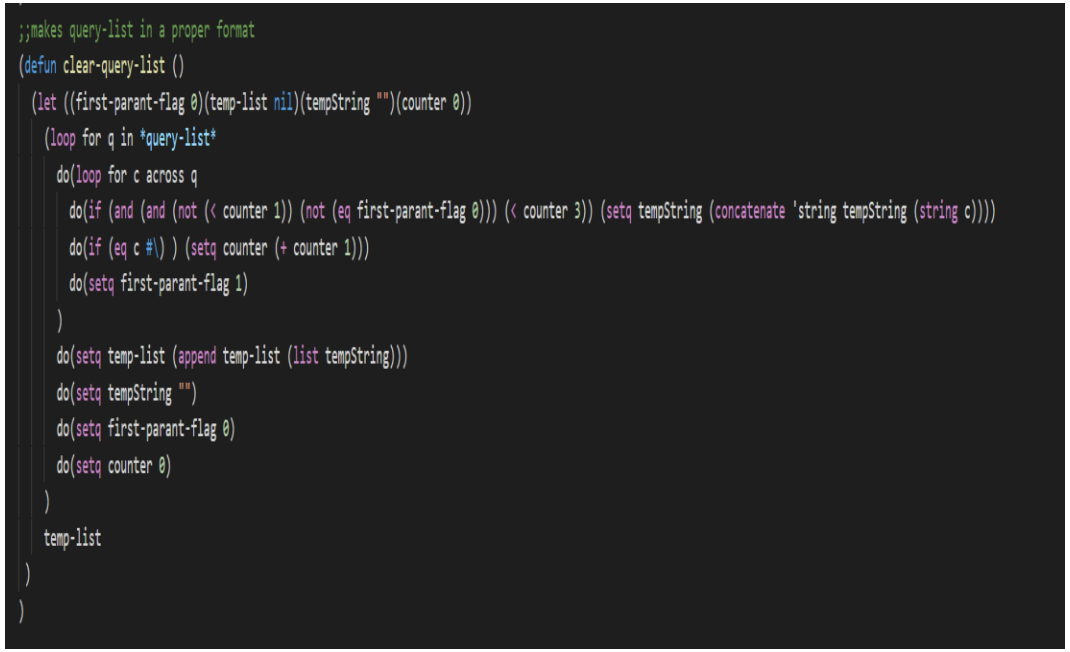
clearList: If there is too much spaces between inputs in file it removes suprus spaces. Lets say there are 5 spaces beetween inputs in file it removes 4 of them.

```lisp
;;if is there a string in another one return it's index
(defun string-include (string1 string2)
  (let* ((string1 (string string1)) (length1 (length string1)))
    (if (zerop length1)
        nil
        (labels ((sub (s)
                   (cond
                    ((> length1 (length s)) nil)
                    ((string= string1 s :end2 (length string1)) string1)
                    (t (sub (subseq s 1))))))
          (sub (string string2))))))

;;categorizes inputs as predicate, fact, or query
(defun categorize-items ()
  (let ((flag 0))
    (loop for x in *line-list*
      do (if (not (equal (string-include "( () (" x) nil)) (setq *query-list* (append *query-list* (list x))))
      do (if (not (equal (string-include ") () )" x) nil)) (setq *fact-list* (append *fact-list* (list x))))
      do (if (and (equal (string-include "( () (" x) nil) (equal (string-include ") () )" x) nil)) (setq *predicate-list* (append *predicate-list* (list x))))
      do (setq flag 0)
    )
  )
)
;;makes fact-list in a proper format
(defun clear-fact-list ()
  (let ((first-parant-flag 0)(temp-list nil)(tempString "")(counter 0))
    (loop for q in *fact-list*
      do(loop for c across q
        do(if (and (not (>= counter 2)) (not (eq first-parant-flag 0))) (setq tempString (concatenate 'string tempString (string c))))
        do(if (eq c #\) ) (setq counter (+ counter 1)))
        do(setq first-parant-flag 1)
      )
      do(setq temp-list (append temp-list (list tempString)))
      do(setq tempString "")
      do(setq first-parant-flag 0)
      do(setq counter 0)
    )
    temp-list
 )
)
```

string-include: checks is there a substring in another string. If there is, then returns it's starting index otherwise nil.

categorize-items: it seperates input as queries, facts, and predicates and append them to assosiative lists as *query-list*, *fact-list*, and *predicate-list*.

clear-fact-list: clears suprus paranthesis from facts. Each fact has a () as tail, so it removes them to ease the calculations.



```
;;makes query-list in a proper format
(defun clear-query-list ()
  (let ((first-parant-flag 0)(temp-list nil)(tempString "")(counter 0))
    (loop for q in *query-list*
      do(loop for c across q
        do(if (and (and (not (< counter 1)) (not (eq first-parant-flag 0))) (< counter 3)) (setq tempString (concatenate 'string tempString (string c))))
        do(if (eq c #\) ) (setq counter (+ counter 1)))
        do(setq first-parant-flag 1)
      )
      do(setq temp-list (append temp-list (list tempString)))
      do(setq tempString "")
      do(setq first-parant-flag 0)
      do(setq counter 0)
    )
    temp-list
  )
)
```

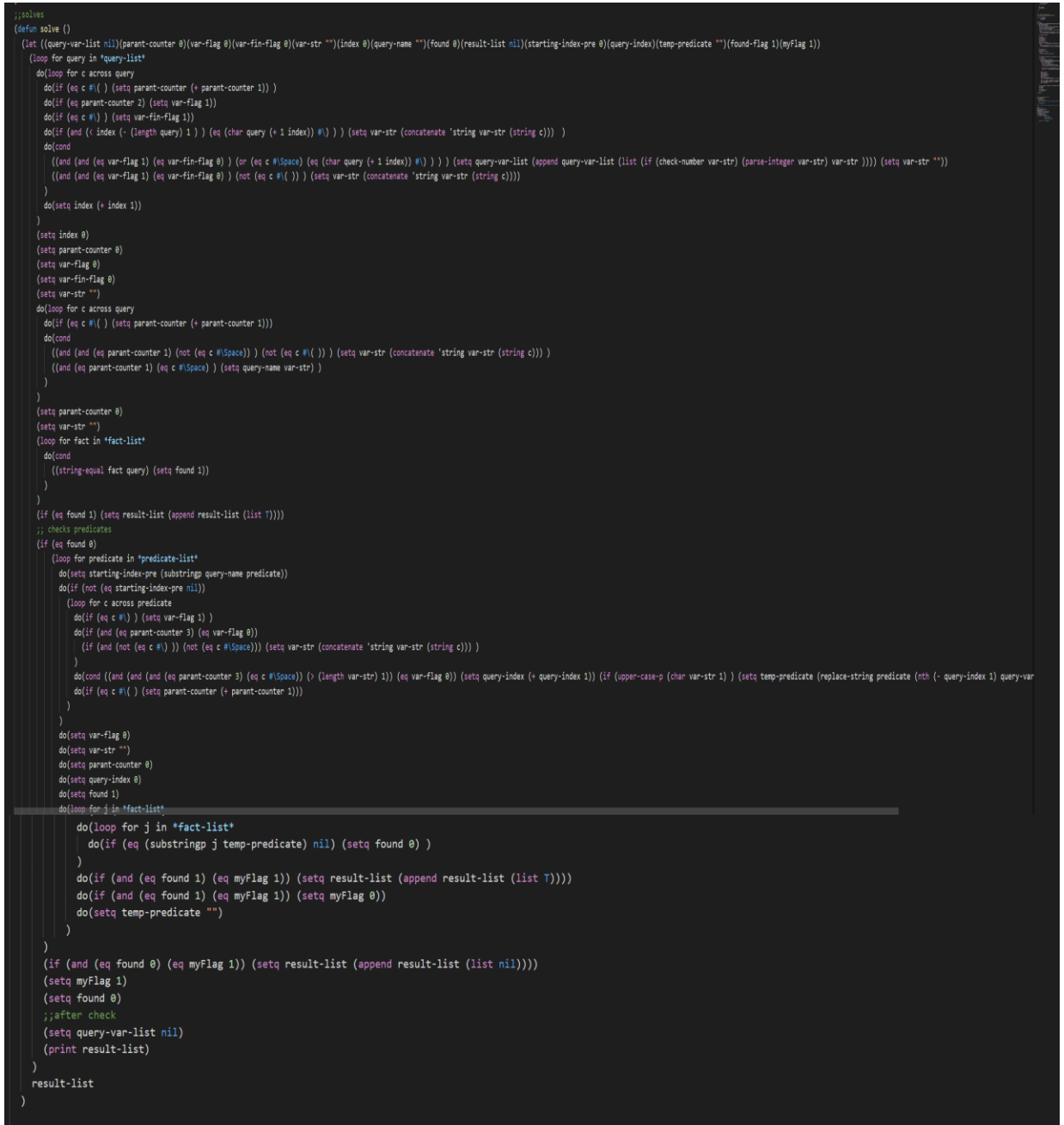
clear-query-list: Clears queries from head part because head parts of the queries are empty lists like fact's tail part.

```lisp
;;replaces a substring in a string
(defun replace-string (str newstr targetstr)
  (let ((starting-index 0) (ending-index 0) (len 0) (index 0) (flag-start 0) (outer-index 0) (found-flag 1) (temp-str ""))
    (loop while (eq found-flag 1)
      do(setq starting-index (substringp targetstr str))
      do(if (not (eq starting-index nil)) (setq found-flag 1) (setq found-flag 0))
      do(if (eq found-flag 1) (setq ending-index (+ starting-index (length targetstr))))
      do(cond ((eq found-flag 1)
        (loop for c across str
          do(cond
            ((< index starting-index) (setq temp-str (concatenate 'string temp-str (string c))))
            ((eq index starting-index) (setq temp-str (concatenate 'string temp-str newstr)))
            ((>= index ending-index) (setq temp-str (concatenate 'string temp-str (string c))))
          )
          do(setq index (+ index 1))
        ) (setq str temp-str)
        )
      )
      do(setq index 0)
      do(setq temp-str "")
    )
    str
  )
)

;;finds if there is a substring in string
(defun substringp (needle haystack &key (test 'char=))

  (search (string needle)
          (string haystack)
          :test test)
)
```

replace-string: This function replaces a substring(targetstr) occurs in string(str) with a spesific string that's given as a argument (newstr)

substringp: checks is there a substring in a string or not

Main function:



```
;;solves
(defun solve ()
  (let ((query-var-list nil)(parant-counter 0)(var-flag 0)(var-fin-flag 0)(var-str "")(index 0)(query-name "")(found 0)(result-list nil)(starting-index-pre 0)(query-index)(temp-predicate "")(found-flag 1)(myFlag 1))
    (loop for query in *query-list*
      do(loop for c across query
        do(if (eq c #\( ) (setq parant-counter (+ parant-counter 1)) )
        do(if (eq parant-counter 2) (setq var-flag 1))
        do(if (eq c #\) ) (setq var-fin-flag 1))
        do(if (and (< index (- (length query) 1) ) (eq (char query (+ 1 index)) #\) ) ) (setq var-str (concatenate 'string var-str (string c)))  )
        do(cond
          ((and (and (eq var-flag 1) (eq var-fin-flag 0) ) (or (eq c #\Space) (eq (char query (+ 1 index)) #\) ) ) ) (setq query-var-list (append query-var-list (list (if (check-number var-str) (parse-integer var-str) var-str )))) (setq var-str ""))
          ((and (and (eq var-flag 1) (eq var-fin-flag 0) ) (not (eq c #\( )) ) (setq var-str (concatenate 'string var-str (string c))))
        )
        do(setq index (+ index 1))
      )
      (setq index 0)
      (setq parant-counter 0)
      (setq var-flag 0)
      (setq var-fin-flag 0)
      (setq var-str "")
      do(loop for c across query
        do(if (eq c #\( ) (setq parant-counter (+ parant-counter 1)))
        do(cond
          ((and (and (eq parant-counter 1) (not (eq c #\Space)) ) (not (eq c #\( )) ) (setq var-str (concatenate 'string var-str (string c))) )
          ((and (eq parant-counter 1) (eq c #\Space) ) (setq query-name var-str) )
        )
      )
      (setq parant-counter 0)
      (setq var-str "")
      (loop for fact in *fact-list*
        do(cond
          ((string-equal fact query) (setq found 1))
        )
      )
      (if (eq found 1) (setq result-list (append result-list (list T))))
      ;; checks predicates
      (if (eq found 0)
          (loop for predicate in *predicate-list*
            do(setq starting-index-pre (substringp query-name predicate))
            do(if (not (eq starting-index-pre nil))
              (loop for c across predicate
                do(if (eq c #\) ) (setq var-flag 1) )
                do(if (and (eq parant-counter 3) (eq var-flag 0))
                  (if (and (not (eq c #\) )) (not (eq c #\Space))) (setq var-str (concatenate 'string var-str (string c))) )
                )
                do(cond ((and (and (and (eq parant-counter 3) (eq c #\Space)) (> (length var-str) 1)) (eq var-flag 0)) (setq query-index (+ query-index 1)) (if (upper-case-p (char var-str 1) ) (setq temp-predicate (replace-string predicate (nth (- query-index 1) query-var
                do(if (eq c #\( ) (setq parant-counter (+ parant-counter 1)))
              )
            )
            do(setq var-flag 0)
            do(setq var-str "")
            do(setq parant-counter 0)
            do(setq query-index 0)
            do(setq found 1)
            do(loop for j in *fact-list*
                do(loop for j in *fact-list*
                  do(if (eq (substringp j temp-predicate) nil) (setq found 0) )
                )
                do(if (and (eq found 1) (eq myFlag 1)) (setq result-list (append result-list (list T))))
                do(if (and (eq found 1) (eq myFlag 1)) (setq myFlag 0))
                do(setq temp-predicate "")
              )
      )
      (if (and (eq found 0) (eq myFlag 1)) (setq result-list (append result-list (list nil))))
      (setq myFlag 1)
      (setq found 0)
      ;;after check
      (setq query-var-list nil)
      (print result-list)
    )
    result-list
  )
```

solve: This function solves the problem.

It firstly gets a query and seperates it into 2 pieces as query-name and query-var-list. Query-name holds name of the query and query-var-list holds the parameters of query. Lets say our query is "legs"("horse" 0). Then query-name will be "legs" and query-var-list will be list of ("horse 0).

After that process it searches facts to check is there any matches between facts and this query. If there is it's directly true.If there is not a match, then it gets into predicate loop. In each step of this loop it gets a predicate from *predicate-list* and it checks is there a variable in this predicate. If there is it replace this variable with associated query paramater throughout all predicate. Lets say our predicate is ;

(("legs" ("X" 0)) (("mammal" ("X")) ("arms" ("X" 0))))

Then if first paramater of query is "horse" like above, it replaces all "X"s with "horse" and it turns into;

(("legs" ("horse" 0)) (("mammal" ("horse")) ("arms" ("horse" 0))))

It applies it all variable in the predicate and after that it checks all paramaters of predicate for matches with facts. If all them are matched then it adds True into result-list otherwise nil.

It applies this method for all predicates and after that it continues with other query and repeat all this process again.

Sample Input and Output Files:

input.txt - Notepad

File Edit Format View Help

```
(
( ("legs" ("X" 0)) ( ("mammal" ("X")) ("arms" ("X" 0)) ) )
( ("legs" ("X" 4)) ( ("mammal" ("X")) ("arms" ("X" 0)) ) )
( ("mammal" ("horse")) () )
( ("arms" ("horse" 0)) () )
( () ("legs" ("horse" 0)) )
( () ("ar" ("ss" 4)) )
( () ("d" (5 6 4)) )
)
```

output.txt - Notepad

File Edit Format View Help

```
(T NIL NIL)
```

Since queries of "ar" and "d" is not satisfies in this facts and predicates they returned NIL(False.), "legs" satisfied so it returned T (True.)

How to run:

```
LHW/Midterm$ clisp 161044065_Demir_Sezer_hw5.lisp
LHW/Midterm$
```

There must be an input file as input.txt at same location with lisp program