Gebze Technical University
Computer Engineering

CSE 341
Programming Languages
2020 Fall

Midterm Project Report

Yusuf Can Kan
161044007

Course Assistant: Muhammet Ali Dede

1 Introduction and Problem Definition

In this problem we will build and interpreter for Prolog, using Common Lisp Programming Languages. We will assume our lexer and parser are already implemented and produced proper results to us. This produced result will be in lisp style. So it provides us to translating data lisp list easly.

We assume that the outcome of the parser will be horn clauses. Each horn clause will have head and body. In our implementation we will build two different function for both of this parts.

Some Important Notes

- Please read the Output Format(Section 4, Page 8) section before testing the code or reading the Tests Cases section.
- It will be better if you use different variable names for different clauses.
- Input list must be in “input.txt”. You can see the predicate result in “output.txt”.
- In pdf file queries does not has an predicate list.
- (() (“legs” (“horse” 4))) This statement must be this => (() ((“legs” (“horse” 4)))) .Please be careful on this.
- In pdf file strings are putted under different characters. Please correct the quote signs. For example; `“legs”` This statement must be this => `"legs"`. Please do directly copy and paste pdf file into the input files.

Implemented / Unimplemented Parts

- I implement all parts of the homework.

In our predicates, objects will start with lowercase letter. If it is starts with lowercase letter it will be taking as variable.

When program starts for the first time it takes all the input queries and sends all queries one by one into the body funcion.

2 Solution Approach

2.1 Body Implementation

The input of body function will be body part of the clause. So it will be predicate list. When our program first starts it takes all the list and sends all of them into this function one by one. When body takes one predicate list, first it sends all predicates in this list into the head function and combines its results.

```
;; body implementation of clause
;; This function takes a body part of clause and proofs it
;; with using given query parameter lists.
; clauseList => list of clauses
; variableList => matched variables  (unification element function from head)
;   between query argumens and arguments
;   from previous body. (UNIFICATION)
; bodyList => list of elements in body. This part will be prooved.
; assignedHead => Constructed variables between query and clause head (headQueryCombine function in head)
(defun body (clauseList variableList bodyList assignedHead)
    (let ((bodyl (assignVariablestoBody variableList bodyList)))
        (if (ifQueryNull (constructBody bodyl clauseList)) ; If one of the body is not prooved
            (let ( ( a (calculateIntersectionElements (constructBody bodyl clauseList) assignedHead)))
                (if (null (cadr a)) nil a))
            nil)
))
```

As you can see this function takes 4 parameter. First one is clauseList (which will be constructed by constructClauseList function), second one is variableList, it contains matched variables between head part of the query (in first call it will be nill), body list which contains all the predicate lists in body part of clause (basically input for body), assignedHead which stores the result variables from head of the clause. This function first calls the assignVariablesToBody function and constructs a new body with assigning variables into the body part. At the same time this part handles the unification of the prolog. After this unification process it proves all the predicates in this body. (In this part it sends every predicate to head function one by one and constructs their results.)

```
(defun constructBody (bodyList clauseList)
    (if (not (null bodyList))
        (append (list (list (cadar bodyList) (calculateHead (car bodyList) clauseList)))
            (constructBody (cdr bodyList) clauseList))
        ))
```

While constructing head results it adds predicates list of parameters to the beginning of the list. This parameters will be helpful in calculating and operations for all of this results.

After this construct process our code checks if is there any false proove in predicate list. If there is it means this proof is wrong. If there is not it calls calculateIntersectionElements function and calculates intersection (and operation) of all this result predicates.

```
(defun calculateIntersectionElements (l headParams)
    (setq result (findAllIntersections l))
    (let ((samevars (findInterElementIndex headParams (car result) 0)))
        (if (null samevars) result
            (append (list headparams) (list (constructHeadVariables headparams samevars (cadr result))))
```

This function basically takes all the calculated proove results from body function and makes comparisons between their parameters. It compares their parameters, finds common elements and constructs a new list with intersection of all this elements with respect to common variables. (At the same time it adds other objecs to the list, in this way we both take intersection with respect to parameters and also store the objects). It returns this new list with adding parameter list to the head of this list.

When we go back our body function, now we have intersection elements. Since this elements construct our results we have our results for the body. Now we can check what is happening inside the head function.

2.2 Head Implementation

In our body implementation we saw that it assignes all the variables taken from input query and makes the unification process. When we look at the head perspective, input of the head will be just one predicate and it tries to match predicate name and predicate elements with head of clause lists.

```
;; finds matched clauses and sends head
(defun calculateHead (query clauseList)
    ;; Finds all proper querys in clause list
    (let ((matches (findForHead query clauseList)))
        (if (null matches) nil ;; if not matched clause return nil.
            (head query matches clauseList))))
```

For the head part, before going to this function, I implemented a wrapper function for this which it first finds the clauses in the clause list that matched with our input query (in this case query is comes from body part). If there is no match in query list it returns false for this input. If there is match it calls head function for all prooving founded clauses.
In head function, it calculates all these clauses list with, matching its input parametes with head (unification) and calling body for its body part. Since here we are changing predicate (which is in body) to the matched clause body we are making unification in here.

```lisp
;; It takes one predicate in body and proofs it.
;; If predicate is fact it returns the predicate variable list
;; If it is not fact, it will send the body part and proof its body first.
(defun head (query matches clauseList)
    (if (not (null matches)) ;; iterate all founded clauses
         (cond
            ((null (cadar matches)) ;If founded clause is fact
                (let ((x (factVariable (cadaar matches) (cadr query))));; Find query result from founded fact
                    (if (null x)
                        (head query (cdr matches) clauseList) ;; If not true
                        ;If clause is true construct list and append to result

                        (append (list (constructHeadResult (reverse (cdr (reverse x))) (cadr query)))
                            (head query (cdr matches) clauseList))))
            )
            (t ;If founded clause is not fact
                (let ((x (unificationElements (cadr query) (cadaar matches)))) ;; variable value match list
                    (if (null x) ;;If clause head does not have any variable

                        (if (null (body clauseList x (cadar matches) (cadr query)));; If body proof is wrong
                            (head query (cdr matches) clauseList)
                            (append (list (cadaar matches)) (head query (cdr matches) clauseList))
                        )
                                    ;;If clause head have variable
                        (progn
                            ;; combine query and head params for body
                            (let ((params (headQueryCombine (cadr query) (cadaar matches))))

                                ;; params stores combination variable for head
                                ;; X 1 Y (head params query) => 3 A B (head params clauese)  ====> 3 1 Y
                                ;; this must go to body

                                (let ((bodyResult (body clauseList x (cadar matches) params)))
                                    (if (null bodyResult)
                                        (head query (cdr matches) clauseList)
                                        (append (cadr bodyResult) (head query (cdr matches) clauseList))
                                ))))))))
```

When we examine this function step by step, we can see that first it controls if matched clause is fact or not,

```lisp
(cond
   ((null (cadar matches)) ;If founded clause is fact
```

If it is fact first it calculates the query result from founded fact. If this result is nil it means clause is not true for our case, it directly checks other matched clauses. If result is not nill it means clause is true, it appends the result proper way to the rest of clause list calculations.

```lisp
(let ((x (factVariable (cadaar matches) (cadr query))));; Find query result from founded fact
    (if (null x)
        (head query (cdr matches) clauseList) ;; If not true
        ;If clause is true construct list and append to result

        (append (list (constructHeadResult (reverse (cdr (reverse x))) (cadr query)))
            (head query (cdr matches) clauseList))))
```

If founded clause is not fact, in this case we need to calculate its body. First we need to find proper values for variables in head part. For this we compare input query and founded clause head and we find the parameter list of clause.

```lisp
(t ;If founded clause is not fact
    (let ((x (unificationElements (cadr query) (cadaar matches)))) ;; variable value match list
        (if (null x) ;;If clause head does not have any variable
```

After that we check it this clause head has any variable or not, if it has not, we calculate the body directly and if result of this is null it means this clause is wrong for out case, if it is true we append the result and calculate rest of the matched clauses.

```
(if (null (body clauseList x (cadar matches) (cadr query)));; If body proof is wrong
    (head query (cdr matches) clauseList)
    (append (list (cadaar matches)) (head query (cdr matches) clauseList))
)
        ;;If clause head have variable
```

If clause head has any variable it means we must find a values for thi variables, first we combine head of input query and head of matched clause for filling variables.

```
;; combine query and head params for body
(let ((params (headQueryCombine (cadr query) (cadaar matches))))

    ;; params stores combination variable for head
    ;; X 1 Y (head params query) => 3 A B (head params clauese)  ====> 3 1 Y
    ;; this must go to body
```

After this process we send all the body to the body function (with doing this we make resolution). In body function it makes resolution process to the body. We also provide params variable to the body for obtaining the parameters that we want.

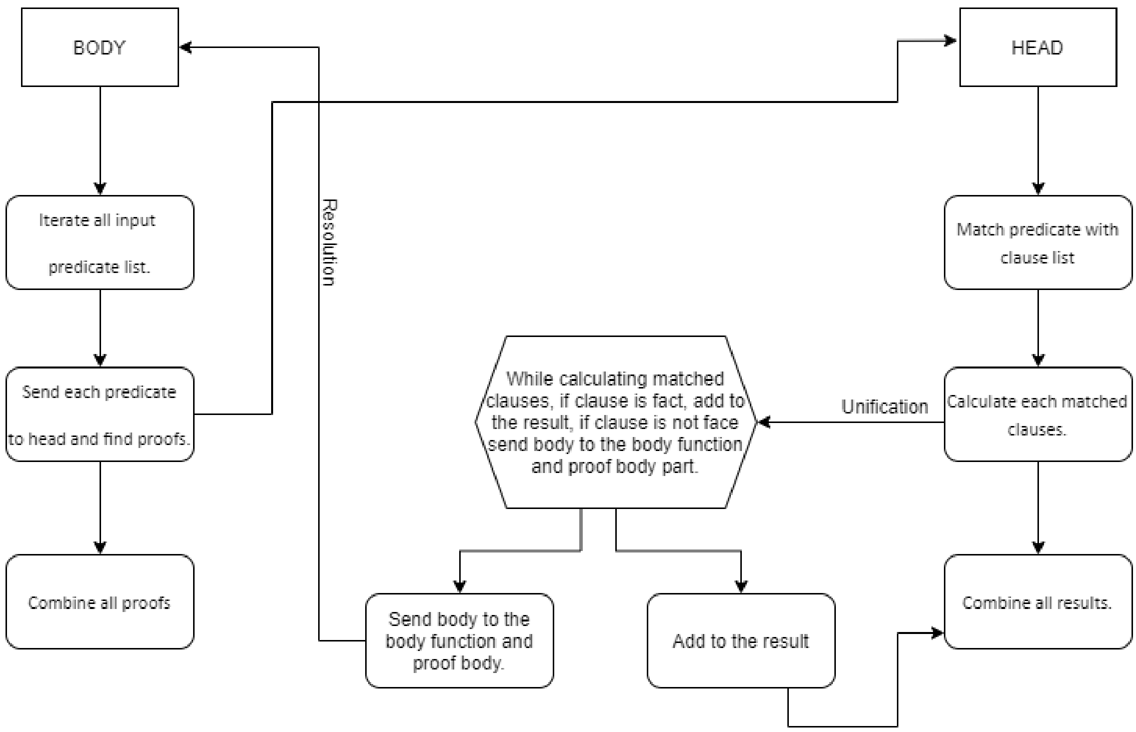
```
(let ((bodyResult (body clauseList x (cadar matches) params)))
    (if (null bodyResult)
        (head query (cdr matches) clauseList)
        (append (cadr bodyResult) (head query (cdr matches) clauseList))
))))))))))
```

At the end we check if result is null or not. If result is null it means our clause is not true, is not match this query so we check the rest of the matched list. If result is not null we add the result to the general result.

3 General Summary of Main Algorithm

In general perspective, both body and head functions are connected to each other in order to provede resolution. When we first call body, it sends all predicate elements to head function and takes results and combines them. When we look at the head function it finds matches with provided predicate and when it finds a match it if its fact it returns true (or variable elements), if it is not fact it sends the body to the body function for calculating (prooving) this part. While doing this body function does unification while assigning variables, head function makes unification with changing predicate head with its matched clause body.

You can examine the main diagram in the next page for understanding working mechanism of body and head functions. more clear view of the algortihm;



<u>Note</u>

All the implementation in this homework done with **<u>recursive programming</u>**.

4 Output Format

- If you provide your query without any variables it just retuns true and false.
- If you provide your query with variables, it returns a list which, first element of this list is a list that represents the variable orders and second element of this list is a list that contains all the possible values for variable order. For example;

  <u>Input:</u> number(1 “X”)
  <u>Output:</u> ((1 X) ((1 2) (1 3) (1 4) (1 5)))

  Input: number(1 “X”), number (“X” 2)
  Output: ((1 X X 2) ((1 1 1 2) (1 5 5 2) (1 4 4 2) (1 8 8 2)))

As you can see here it combines predicate lists all variables together.

Each line in the output.txt file represents different query result. (Some result list might be written in 2 line).

If output list contains parameter (Such as A B C) it means this parameter value does not matter. We can give any value to that parameter.

5 Test Cases

5.1 Test1

5.1.1 input.txt File

```
(
    ( ("ancestor" ("X" "Y")) ( ("father" ("X" "A")) ("mother" ("A" "Y")) ))
    ( ("ancestor" ("X" "Y")) ( ("father" ("X" "Y")) ))
    ( ("ancestor" ("X" "Y")) ( ("mother" ("X" "Y")) ))
    ( ("sibling" ("X" "Y")) ( ("mother" ("A" "X")) ("mother" ("A" "Y")) ))
    ( ("sibling" ("X" "Y")) ( ("father" ("A" "X")) ("father" ("A" "Y")) ))
    ( ("married" ("X" "Y")) ( ("father" ("X" "A")) ("mother" ("Y" "A")) ))
    ( ("father" ("bill" "tom")) ())
    ( ("father" ("bill" "pam")) ())
    ( ("father" ("mike" "sam")) ())
    ( ("father" ("alan" "rachel")) ())
    ( ("father" ("mike" "monika")) ())
    ( ("father" ("john" "emily")) ())
    ( ("mother" ("emily" "tom")) ())
    ( ("mother" ("emily" "rachel")) ())
    ( ("mother" ("amelia" "mike")) ())
    ( ("mother" ("amelia" "emily")) ())
    ( ("mother" ("ava" "harry")) ())
    ( ("mother" ("joanne" "pam")) ())
    ( ("mother" ("joanne" "tom")) ())

    ( () (("mother" ("emily" "tom")) ) )
    ( () (("mother" ("emily" "mike")) ) )
    ( () (("ancestor" ("amelia" "mike")) ) )
    ( () (("ancestor" ("john" "tom")) ) )
    ( () (("ancestor" ("X" "Y")) ) )
    ( () (("sibling" ("amelia" "Y")) ) )
    ( () (("sibling" ("pam" "tom")) ) )
    ( () (("married" ("pam" "tom")) ) )
    ( () (("married" ("X" "Y")) ) )
)
```

5.1.2 output.txt File

```
True
False
True
True
((X Y)
 ((bill tom) (bill pam) (mike sam) (alan rachel) (mike monika) (john emily)
  (emily tom) (emily rachel) (amelia mike) (amelia emily) (ava harry)
  (joanne pam) (joanne tom) (john tom) (john rachel)))
False
True
False
((X Y) ((bill emily) (bill joanne) (bill joanne) (alan emily) (john amelia)))
```

As yu can see in this test first 4 clause, there is no variable so it must return true and false. In third and forth cases there must be resolution operation must be done. You can check this case with looking this results.

In 5th query we want all the possible pairs with has ancestor relationship with each other. It produced list of all possible pairs. (Also this case need resolution.)

Last query is also the same but this time it looks for married relationship. If you look at the married relationship it takes pairs with mother and father relationship which has same children. You can check it also from clause list.

5.2 Test 2

5.2.1 input.txt File

```
(
    ( ("friend" ("ali" "veli")) ())
    ( ("friend" ("veli" "semih")) ())
    ( ("friend" ("yusuf" "ahmet")) ())
    ( ("friend" ("yusuf" "cemal")) ())
    ( ("friend" ("ahmet" "cemal")) ())
    ( ("bestfriend" ("X" "Z")) ( ("friend" ("X" "Y")) ("friend" ("Y" "Z"))))

    ( () (("friend" ("ali" "veli")) ))
    ( () (("bestfriend" ("A" "B")) ))
)
```

5.2.2 output.txt File

```
True
((A B) ((ali semih) (yusuf cemal)))
```

As you can see in this test ali is friend with veli so it returns true and variables A and B cam be prooved with 2 different way.
So variable

- A B = ali semih
  Or
- A B = yusuf cemal

5.3 Test 3

5.3.1 input.txt File

```
(
    ( ("food" ("burger")) ())
    ( ("food" ("sandwich")) ())
    ( ("food" ("pizza")) ())
    ( ("lunch" ("sandwich")) ())
    ( ("dinner" ("pizza")) ())

    ( ("meal" ("X")) ( ("food" ("X")) ))

    ( () ( ("meal" ("X")) ))
    ( () (("dinner" ("sandwich")) ))

)
```

5.3.2 output.txt File

```
((X) ((burger) (sandwich) (pizza)))
False
```

In this case first query wants all the meals. Ans second one just checks if sandwich is dinner or not. This test written only for showing code works on simple cases.

5.4 Test 4

5.4.1 input.txt File

```
(
    ( ("legs" ("X" 2)) ( ("mammal" ("X")) ("arms" ("X" 2)) ) )
    ( ("legs" ("X" 4)) ( ("mammal" ("X")) ("arms" ("X" 0)) ) )
    ( ("mammal" ("horse")) () )
    ( ("arms" ("horse" 0)) () )
    ( ("gebze" ("ali" 1)) () )
    ( ("gebze" ("horse" 0)) () )
    ( ("gebze" ("yy" 2)) () )
    ( ("gebze" ("veli" 0)) () )
    ( ("gebze" ("horse" 3)) () )
    ( ("gebze" ("h2" 3)) () )
    ( ("gebze" ("h4" 3)) () )
    ( ("gebze" ("h3" 3)) () )
    ( ("gebze" ("h5" 3)) () )
    ( ("gebze" ("h5" 3)) ("gebze" ("horse" 3))  )
    ( ("gebze" ("world" 0)) () )
    ( ("gebze" ("world" 5)) () )
    ( ("gebze" ("world" 5)) () )
    ( ("gebze" ("world" 6)) () )
    ( ("gebze" ("world" 7)) () )
    ( ("gebze" (7 7)) () )

    ( () ("legs" ("horse" 4)) )
    ( () ("mammal" ("horse")) )
    ( () (("mammal" ("gebze")) ))
    ( () (("gebze"  ("X" "Y"))))
)
```

5.4.2 output.txt File

```
True
True
False
((X Y)
 ((ali 1) (horse 0) (yy 2) (veli 0) (horse 3) (h2 3) (h4 3) (h3 3) (h5 3)
  (world 0) (world 5) (world 5) (world 6) (world 7) (7 7) (h5 3)))
```

First and second nad third query just wants the proof of given parameters. As you can see there is no mammal(gebze) fact(clause) in clause list. So third query is wrong.

In forth query it gives all the possible X and Y parameters for gebze relationship.

5.5 Test 5

5.5.1 input.txt File

Clause List for test5 ;

```
(
    ( ("evennumber" (0)) ())
    ( ("evennumber" (2)) ())
    ( ("evennumber" (4)) ())
    ( ("evennumber" (6)) ())
    ( ("evennumber" (8)) ())
    ( ("twodigit" ("X" 0)) ())
    ( ("twodigit" ("X" 1)) ())
    ( ("twodigit" ("X" 2)) ())
    ( ("twodigit" ("X" 3)) ())
    ( ("twodigit" ("X" 4)) ())
    ( ("twodigit" ("X" 5)) ())
    ( ("twodigit" ("X" 6)) ())
    ( ("twodigit" ("X" 7)) ())
    ( ("twodigit" ("X" 8)) ())
    ( ("twodigit" ("X" 9)) ())
    ( ("threedigit" ("X" "Y" 0)) ())
    ( ("threedigit" ("X" "Y" 1)) ())
    ( ("threedigit" ("X" "Y" 2)) ())
    ( ("threedigit" ("X" "Y" 3)) ())
    ( ("threedigit" ("X" "Y" 4)) ())
    ( ("threedigit" ("X" "Y" 5)) ())
    ( ("threedigit" ("X" "Y" 6)) ())
    ( ("threedigit" ("X" "Y" 7)) ())
    ( ("threedigit" ("X" "Y" 8)) ())
    ( ("threedigit" ("X" "Y" 9)) ())
    ( ("twodigiteven" ("X" "Y")) ( ("evennumber" ("Y")) ("twodigit" ("X" "Y")) ))
    ( ("threedigiteven" ("X" "Y" "Z")) ( ("evennumber" ("Z")) ("threedigit" ("X" "Y" "Z")) ))
```

Query List for test5 ;

```
( () (("twodigiteven" ("A" "B")) ))
( () (("twodigiteven" (1 2)) ))
( () (("twodigiteven" (8 "X")) ))
( () (("twodigiteven" (4 5)) ))
( () (("twodigiteven" (3 8)) ))
( () (("threedigiteven" ("A" "B" "C")) ))
( () (("threedigiteven" ("A" "B" 1)) ))
( () (("threedigiteven" ("A" "B" 2)) ))
( () (("threedigiteven" ("A" "B" 3)) ))
( () (("threedigiteven" (5 2 9)) ))
```

5.5.2 output.txt File

```
((A B) ((A 0) (A 2) (A 4) (A 6) (A 8)))
True
((8 X) ((8 0) (8 2) (8 4) (8 6) (8 8)))
False
True
((A B C) ((A B 0) (A B 2) (A B 4) (A B 6) (A B 8)))
False
((A B 2) ((A B 2)))
False
False
```
,

In this example I defined facts for even numbers and two and three digit numbers. Twodigiteven predicate finds all the possible 2 digit numbers and combine them with even predicate. So in first query we want all possible parameters for this query. It return A 0 - A 2 - A 4 ... In this case A represents that this parameter can take any value because it defined in this way.

First 5 query looks different examples of twodigiteven case. Last 5 query checks the similar situations but for defined three digit case.

Test6 test is in next page..

5.6 Test 6

5.6.1 input.txt File

```
(
    ( ("number" (1 2)) ())
    ( ("number" (1 3)) ())
    ( ("number" (1 6)) ())
    ( ("number" (1 8)) ())

    ( ("number" (2 0)) ())
    ( ("number" (2 2)) ())
    ( ("number" (2 3)) ())
    ( ("number" (2 5)) ())
    ( ("number" (2 7)) ())
    ( ("number" (2 8)) ())
    ( ("number" (2 9)) ())

    ( ("number" (3 5)) ())
    ( ("number" (3 9)) ())
    ( ("number" (3 0)) ())
    ( ("number" (3 8)) ())
    ( ("number" (3 1)) ())
    ( ("number" (3 2)) ())

    ( ("number" (7 5)) ())
    ( ("number" (8 5)) ())


    ( () (("number" (2 "X")) ))
    ( () (("number" (2 5)) ))
    ( () (("number" ("X" 5)) ))

    ( () (("number" (2 "X")) ("number" ("X" 5)) ))
)
```

5.6.2 output.txt File

```
((2 X) ((2 0) (2 2) (2 3) (2 5) (2 7) (2 8) (2 9)))
True
((X 5) ((2 5) (3 5) (7 5) (8 5)))
((2 X X 5) ((2 2 2 5) (2 3 3 5) (2 7 7 5) (2 8 8 5)))
```

In this case I defined simple facts with combined with numbers. First and third case just looks for possible variables, second case profs is 2 5 defined in our clause list.

In last case we have and relationships. In order to define this predicate list with different head, I wanted to test directly providing the program. It takes 2 X and X 5 cases and tries to proof it. As you can see it combined 2 X and X 5 together and construct 2 X X 5 parameter and fill the result list with respect to this parameters. You can see that colums of 2 and 5 (first and last emement of results) are fixed and second and third elements are the same.(second and third is same because parameter defined as X X),