

HW5 – MIDTERM PROJECT

Can Duyar - 171044075

1. Description

This is a Lisp project simulates a Prolog compiler which input is a clauses list. The list is the output of a lexer and a parser which takes a Prolog program and generate a data structure in Lisp.

The outcome of the parser is a list of horn clauses.

Each horn clause is a list of two entries, namely head and body.

- Head part is itself a list. If the clause is a query, head will be nil. Otherwise, it will be a predicate.
- Body is also a list with zero more entries. If the clause is a fact, the body will be nil. Otherwise, it will be a list of predicates.

. A predicate is a list with two entries: (predicate_name list_of_parameters).

The first entry is a string indicating the name of the predicate.

The second is a list of (possibly empty) parameters. The list of parameters can have string or numeric entries.

For example, the following code:

```
legs(X,2) :- mammal(X),arms(X,2).
```

```
legs(X,4) :- mammal(X),arms(X,0).
```

```
mammal(horse).
```

```
arms(horse,0).
```

```
?- legs(horse,4).
```

will generate the list as :

```
(  
  ( ("legs" ("X" 2)) ( ("mammal" ("X")) ("arms"  
    ("X" 2)) ) ) ( ("legs" ("X" 4)) ( ("mammal"  
    ("X")) ("arms" ("X" 0)) ) )  
  ( ("mammal" ("horse")) ( ) )  
  ( ("arms" ("horse" 0)) ( ) )  
  ( ( ) ("legs" ("horse" 4)) )  
)
```

And the output of this project is "true" as Prolog does.

2. Program design

a. Main function:

```
(defun run (program)  
  ...  
)
```

This is the entry point of the program uses inputs of the input.txt a list of Horn clauses and then output the result of query to print on the output.txt.

```
;; Main program  
(defun run (program)  
  ;; Get a list of query, facts and rules  
  (let ((query (get-query program))  
        (facts (get-facts program))  
        (rules (get-rules program))  
        )  
    (let ((result (solve (get-matched-rules query rules) query facts rules)))  
      ;; If the list of values less than the required params in query, fail  
      ;; If they're equal, just keep the variable (remove the constant)  
      ;; If all prams in query is constant, return true ( because empty list is for failure)  
      (if (< (length result ) (length (second query))) (list "False")  
          (let ((data (show-result (print result))))  
            (cond  
              ((data data)  
               ((check-true-or-false result) (list "True"))  
               (T (list "False"))  
              ))  
          )  
      )  
    )  
)
```

b. Utilities

i. (defun member-list (E L)

...

)

Check if an element E is member of the list

L or not

ii. (defun append-list (L1 L2)

...

)

Append a list L1 to list L2 and return the appended list

iii. (defun append-member (E L)

...

)

Add element E to list L if E is not member of L

iv. (defun is-variable? (term)

...

)

Check if a term is variable (such as X,Y,Z) or constant

```
;; List helper functions
(defun append-member (E L)
  (if (member-list E L) L (append E L))
)
(defun append-list (L1 L2)
  (if (not L1) L2
      (append-member (car L1) (append-list (cdr L1) L2)))
)

(defun member-list (E L)
  (if (not L) nil
      (if (not (equal E (car L))) (member-list E (cdr L)) t))
)
```

c. Algorithm:

There are several ways to implement this program. The easiest way is to use recursion.

The following is the step by step:

i. extract query from the list. The query is a horn clause which has head is empty

ii. extract facts from the list. The facts are horn clauses which have body is empty

iii. extract the rules which LHS (left hand side) predicate and RHS predicates

iv. for each rule

- If resolving the rule can answer the query, stop and show output
- If not, go to next rule
- For each rule:
 - + Go to the step (iv) for each RHS predicate.

If all RHS predicates don't conflict during resolving and the query can be answer, stop and show output.

For each rule is resolved, assigning the value to variables in that rules.

* All parts are working correctly and when you write queries one by one, you have all answers about input.txt (please ask just one question for each running.)

3-Testing

Test-1:

i. Program

```
legs(X,2) :- mammal(X),arms(X,2).
```

```
legs(X,4) :- mammal(X),arms(X,0).
```

```
mammal(horse).
```

```
arms(horse,0).
```

```
?- legs(horse,4).
```

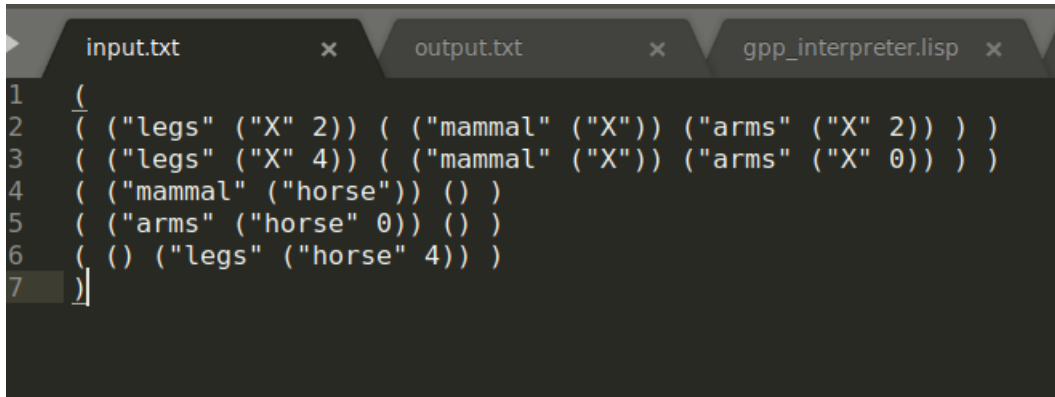
ii. Horn list:

```
(  
  ( ("legs" ("X" 2)) ( ("mammal" ("X")) ("arms" ("X" 2)) ) )  
  ( ("legs" ("X" 4)) ( ("mammal" ("X")) ("arms" ("X" 0)) ) )  
  ( ("mammal" ("horse")) ( ) )  
  ( ("arms" ("horse" 0)) ( ) )  
  ( ( ) ("legs" ("horse" 4)) )  
)
```

iii) Output:

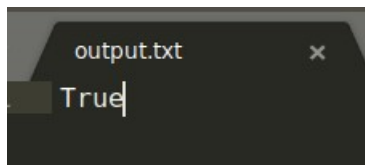
True

input.txt:



```
1  (  
2  ( ("legs" ("X" 2)) ( ("mammal" ("X")) ("arms" ("X" 2)) ) )  
3  ( ("legs" ("X" 4)) ( ("mammal" ("X")) ("arms" ("X" 0)) ) )  
4  ( ("mammal" ("horse")) ) )  
5  ( ("arms" ("horse" 0)) ) )  
6  ( () ("legs" ("horse" 4)) )  
7  )
```

output.txt:



```
output.txt  
True
```

Test-2:

i. Program

legs(X,2) :- mammal(X),arms(X,2).

legs(X,4) :- mammal(X),arms(X,0).

mammal(horse).

arms(horse,0).

?- legs(horse,2).

ii. Horn list:

```
(  
  ( ("legs" ("X" 2)) ( ("mammal" ("X")) ("arms" ("X" 2)) ) )  
  ( ("legs" ("X" 4)) ( ("mammal" ("X")) ("arms" ("X" 0)) ) )  
  ( ("mammal" ("horse")) ( ) )  
  ( ("arms" ("horse" 0)) ( ) )  
  ( ( ) ("legs" ("horse" 2)) )  
)
```

iii. Output:

False

input.txt:

```
input.txt  output.txt  gpp_interpreter.lisp
1  (
2  ( ("legs" ("X" 2)) ( ("mammal" ("X")) ("arms" ("X" 2)) ) )
3  ( ("legs" ("X" 4)) ( ("mammal" ("X")) ("arms" ("X" 0)) ) )
4  ( ("mammal" ("horse")) () )
5  ( ("arms" ("horse" 0)) () )
6  ( () ("legs" ("horse" 2)) )
7  )
```

output.txt:

```
output.txt
1  False
```

Test-3:

i. Program

legs(X,2) :- mammal(X),arms(X,2).

legs(X,4) :- mammal(X),arms(X,0).

mammal(horse).

arms(horse,0).

?- legs(X,Y).

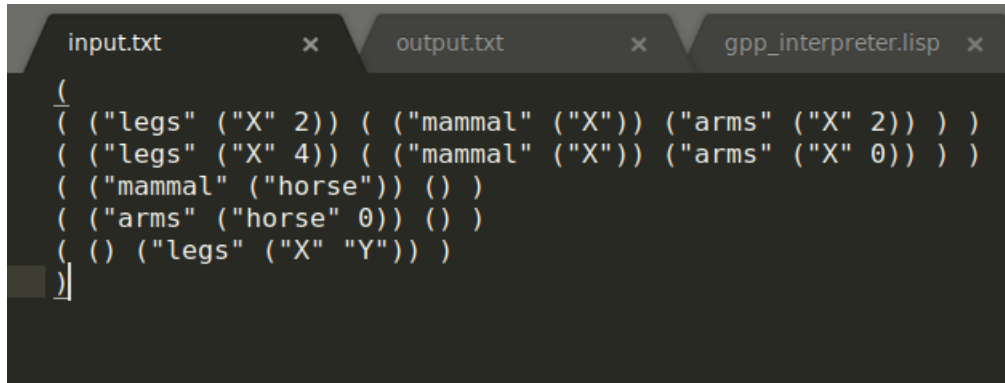
ii. Horn list:

```
(
  ( ("legs" ("X" 2)) ( ("mammal" ("X")) ("arms" ("X" 2)) ) )
  ( ("legs" ("X" 4)) ( ("mammal" ("X")) ("arms" ("X" 0)) ) )
  ( ("mammal" ("horse")) () )
  ( ("arms" ("horse" 0)) () )
  ( () ("legs" ("X" "Y")) )
)
```


iii. Output:
(X horse)
(Y 4)

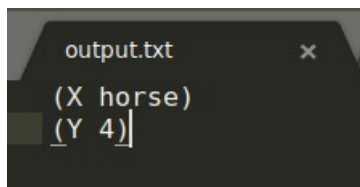
it means that X is horse, Y is 4.

input.txt:



```
(  
  ( ("legs" ("X" 2)) ( ("mammal" ("X")) ("arms" ("X" 2)) ) )  
  ( ("legs" ("X" 4)) ( ("mammal" ("X")) ("arms" ("X" 0)) ) )  
  ( ("mammal" ("horse")) ( ) )  
  ( ("arms" ("horse" 0)) ( ) )  
  ( ( ) ("legs" ("X" "Y")) )  
)
```

output.txt:



```
(X horse)  
(Y 4)
```

Test-4:

i. Program

legs(X,2) :- mammal(X),arms(X,2).

legs(X,4) :- mammal(X),arms(X,0).

mammal(horse).

mammal(human).

arms(human,2).

arms(horse,0).

?- legs(X,2).

ii. Horn list:

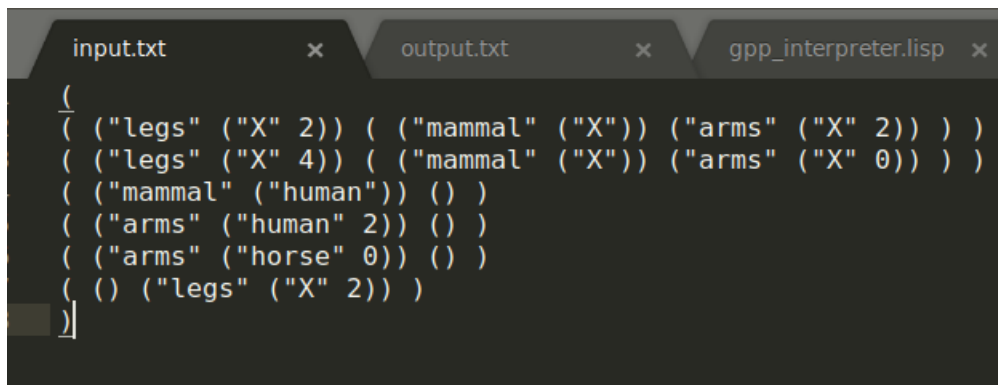
```
(  
  ( ("legs" ("X" 2)) ( ("mammal" ("X")) ("arms" ("X" 2)) ) )  
  ( ("legs" ("X" 4)) ( ("mammal" ("X")) ("arms" ("X" 0)) ) )  
  ( ("mammal" ("human")) () )  
  ( ("arms" ("human" 2)) () )  
  ( ("arms" ("horse" 0)) () )  
  ( () ("legs" ("X" 2)) )  
)
```

iii. Output:

(X human)

It means that X is human.

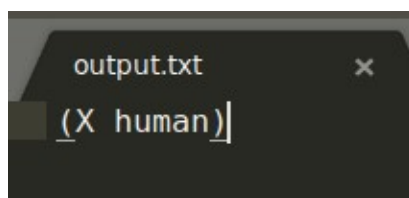
Input.txt:



The screenshot shows a text editor with three tabs: 'input.txt', 'output.txt', and 'gpp_interpreter.lisp'. The 'input.txt' tab is active and displays the following text:

```
(  
  ( ("legs" ("X" 2)) ( ("mammal" ("X")) ("arms" ("X" 2)) ) )  
  ( ("legs" ("X" 4)) ( ("mammal" ("X")) ("arms" ("X" 0)) ) )  
  ( ("mammal" ("human")) () )  
  ( ("arms" ("human" 2)) () )  
  ( ("arms" ("horse" 0)) () )  
  ( () ("legs" ("X" 2)) )  
)
```

Output.txt:

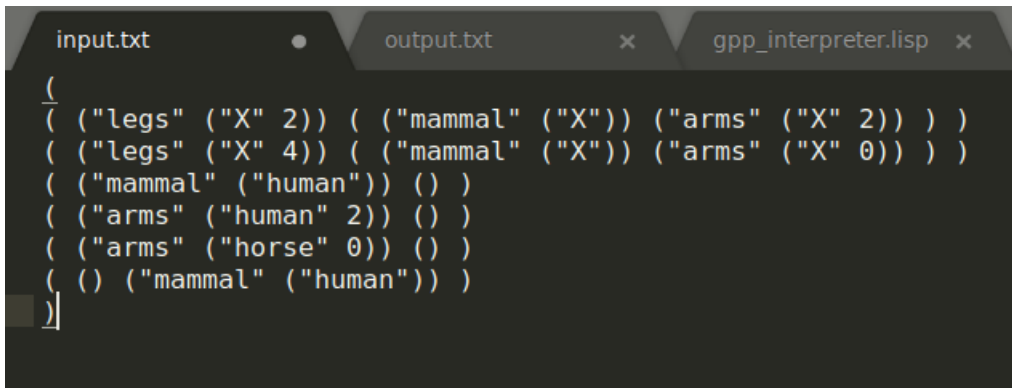


The screenshot shows a text editor with one tab: 'output.txt'. The 'output.txt' tab is active and displays the following text:

```
(X human)
```

You can test it with different queries, if you want.

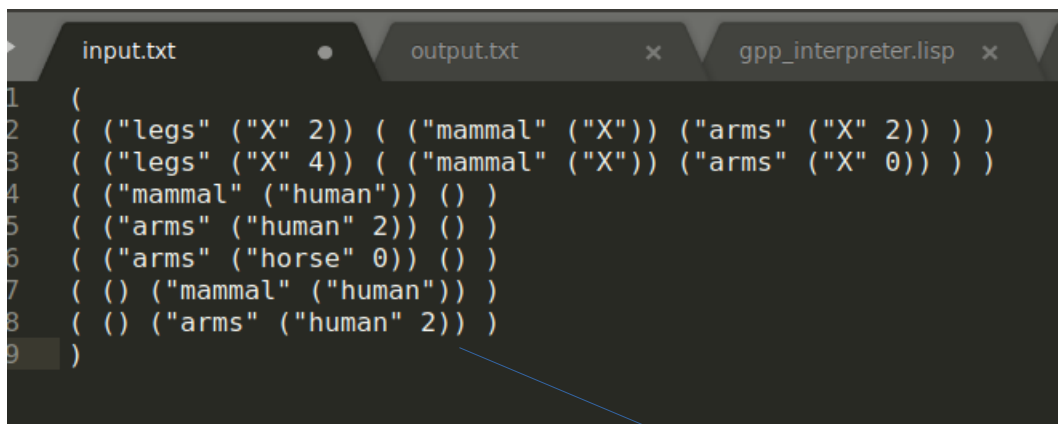
Please ask step by step: Let's suppose that you have input.txt that has



```
input.txt      output.txt      gpp_interpreter.lisp
(
  ( ("legs" ("X" 2)) ( ("mammal" ("X")) ("arms" ("X" 2)) ) )
  ( ("legs" ("X" 4)) ( ("mammal" ("X")) ("arms" ("X" 0)) ) )
  ( ("mammal" ("human")) () )
  ( ("arms" ("human" 2)) () )
  ( ("arms" ("horse" 0)) () )
  ( () ("mammal" ("human")) )
)
```

then it returns false because we know that human is a mammal in our code.

It works properly but you can not ask more than one question at the same time (like example in the below). Please use same input.txt and ask one by one if you want to have all queries' answers. Program works correctly...



```
input.txt      output.txt      gpp_interpreter.lisp
1  (
2    ( ("legs" ("X" 2)) ( ("mammal" ("X")) ("arms" ("X" 2)) ) )
3    ( ("legs" ("X" 4)) ( ("mammal" ("X")) ("arms" ("X" 0)) ) )
4    ( ("mammal" ("human")) () )
5    ( ("arms" ("human" 2)) () )
6    ( ("arms" ("horse" 0)) () )
7    ( () ("mammal" ("human")) )
8    ( () ("arms" ("human" 2)) )
9  )
```

Don't use second one.

Note: My code only works with normal quotation marks (" "), doesn't work with (" "). Please consider this when you test it!