Description

This is a Lisp project simulates a Prolog program which input is a special form horn clauses lisp list. The output of program is list of answers to queries. Each input horn clause is a list of two entries, namely head and body.

Input file syntax must meet the following conditions:

- All horn clauses are in list
- Head part is itself a lisp 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 content of input.txt file:

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

Is represents the next prolog program equivalent:

```
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 program is "yes" as Prolog does.

Program design

Main function

```
(defun main ()
...
)
```

This is the entry point of the program which reads input.txt file a list of Horn clauses and then output the result of queries output.txt file and collected knowledge (horn clauses list). The content of file “input.txt” will be converted to lisp list by built in function read-from-string. Then starts function which processes input list item by item recursively.

Input processing

The main input processing function is process-input which defined as.

```
(defun process-input (input-list knowledge output)
...
)
```

The function takes 3 arguments:

- input-list - rest list of horn clauses which was read from input.txt
- knowledge - collected horn clauses which are not query
- output - the “yes” or “no” answers to queries

It returns list (knowledge, output) where knowledge is collected horn clauses which are not query and output is the answers to queries as list of strings like “?-query1(p1 p2 p3) => yes” and “?-query1(p1 p2 p3) => no”. The algorithm of function is:

1. if list is empty return collected knowledge and output lists
2. else take first element of input-list
3. if it’s a query, then call function process-query and collects its output in output list
4. if it’s not a query then add it to knowledge list
5. then call process-input on the rest of input-list and collected knowledge and output

Query processing

The main function to processing queries is process-query.

```
(defun process-query (query knowledge)
...
)
```

It takes 2 arguments

- query - the second item of horn clause (nil predicate)
- knowledge - collected non query horn clauses

The algorithm:

1. if query is list of predicates then call collect function for each item
2. if query is predicate then call collect for query
3. then if result is not nil, then convert query to string and add to string “yes”
4. else, convert query to string and add to string “no”

Collecting a query variable

The collect function defined as

```
(defun collect(predicate knowledge)
...
)
```

Selects horn clauses which matches predicate.

The algorithm of collect function:

1. Select all horn clauses which match given predicate from knowledge
2. If selected horn clauses is not empty, then for each matched horn clause
   a. If horn clause is a simple fact, then return true
   b. Else call function collect-values which collects all depend predicate values and joins them

Collecting depend predicate values

The function collect-values collects depended predicate values. It defined as

```
(collect-values (args values entry knowledge)
...
)
```

The algorithm

1. For each depended predicate recursively runs the function collect

2. Join the result sets of collected variables values by variable name

Testing

1. Input

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

Output

?-legs(horse 4) => yes

2. Input

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

Output

?-legs(horse 2) => no

3. Input

```
 (
( ("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" ("human" 2)) )
)
```

Output

```
?=>legs(human 2) => yes
```