Synopsis Lab 4

You will learn how pattern matching works and you will meet ELIZA, an ancestor of modern bots.

One task often encountered in Computer Science is matching objects and patterns and make decisions accordingly. For example, many rule-based system rely on automatically matching rules over known facts and taking actions accordingly (e.g., draw conclusions and add them at the set of known facts).

The function `match` in `patternMatching.lsp` implements the mechanism of pattern matching. It relies on macros, so we will leave the implementation details aside and focus on its functionality. This function has 3 arguments: a pattern `p`, a datum `d` and a list of `assignmnets` storing the variable bindings which might occur (e.g. the pattern or the datum contains some variables). If the matching process fails, the result returned will be `NIL`; otherwise, the result is either `T` or a non-empty list containing variable bindings. The cases considered and the returned value for each of them are:

- **identical:** `p` and `d` are identical and `assignments` is empty. Thus `match` should return `T`:

  ```
  >(match '(color apple red) '(color apple red) NIL)
  T
  ```

- **identical, non-empty list:** `p` and `d` are identical and `assignments` is not empty. Now `match` should return `assignments`:

  ```
  >(match '(color apple red) '(color apple red) '((shape round)))
  ((SHAPE ROUND))
  ```

- **different:** `p` and `d` are different. In this case `match` should return `NIL`:

  ```
  >(match '(color apple green) '(color apple red) NIL)
  NIL
  ```

- **singleton wildcard:** pattern `p` contains the special symbol `?`; this is a like a wildcard which matches one symbol in `d`. In this case `match` should return `T`:

  ```
  >(match '(color ? red) '(color apple red) NIL)
  T
  ```

- **multiple wildcard:** pattern `p` contains the special symbol `+`; this is like a wildcard which matches one or more symbols in `d`. In this case `match` should return `T`:

  ```
  >(match '(color +) '(color apple red) NIL)
  T
  ```

- **instantiate variable:** pattern `p` contains the special matching expression `(> x)`; this designates a variable `x` which will match a symbol in `d` **and** will associate `x` with that symbol in the `assignments` list. In this case `match` should return the new value of `assignments`:

```
>(match '(color (> fruit) red) '(color apple red) NIL)
((FRUIT APPLE))
```

- **instantiate list:** pattern `p` contains the special matching expression `(+ l)`; this designates a list `l` which will match one or more symbols in `d` **and** will associate `l` with that list of symbols in the `assignments` list. In this case `match` should return the new value of `assignments`:

```
>(match '(color (+ result)) '(color apple red) NIL)
((RESULT (APPLE RED)))
```

- **already instantiated variable:** pattern `p` contains the special matching expression `(< x)`; this means the value attached to variable `x` inside the list `assignments` must be the same as the corresponding symbol or list of symbols in `d`. If this is not the case, `match` should return `NIL`:

```
>(match '(color (< fruit) red) '(color apple red) '( (fruit apple)))
((FRUIT APPLE))  ; success
>(match '(color (< fruit) red) '(color apple red) '( (fruit peach)))
NIL
```

- **restricted symbol:** pattern `p` contains the special matching expression

  `(RESTRICT ? pred1 ... predn)`; this means the predicates `pred1,...,predn` must hold for the corresponding symbols in `d`. If this is not the case, `match` should return `NIL`:

```
>(DEFUN colorp (c) (MEMBER c '(red green)))
COLORP
>(match '(color apple (RESTRICT ? colorp)) '(color apple red) NIL)
T
>(match '(color apple (RESTRICT ? colorp)) '(color apple yellow) NIL)
NIL
```

Now we'll see how the pattern matching mechanism can be applied to develop a basic bot. Look in `eliza.lsp`; you will find a psychiatrist emulating program which is able to have a short conversation with you. Load the `goEliza.lsp` file and call the wrapper function `(go-Eliza)`. The function `doctor` simply matches your utterances over pre-defined patterns and chooses the replies accordingly. A typical conversation, including both the user's and ELIZA's replies, would be:

```
>(go-eliza)
Loading patternMatching.lsp
Finished loading patternMatching.lsp
Loading eliza.lsp
Finished loading eliza.lsp
```

```
(SPEAK UP!)
(I AM WORRIED ABOUT THE WEATHER)

(HOW LONG HAVE YOU BEEN WORRIED ABOUT THE WEATHER)
(SINCE MY MOTHER SPOKE)

(TELL ME MORE ABOUT YOUR FAMILY)
(THEY ARE ALL INTO COMPUTERS THESE DAYS)

(DO MACHINES FRIGHTEN YOU)
(NO)

(PLEASE DO NOT BE SO SHORT WITH ME)
(GO TO HELL DARN IT)

(PLEASE DO NOT USE WORDS LIKE THAT)
(YOU ARE FRUSTRATING ME)

(EARLIER YOU SPOKE OF YOUR MOTHER)
(I WOULD LIKE TO TALK ABOUT MY PROBLEM NOW)

(I AM SORRY OUR TIME IS UP)
NIL
```

Take a look at the function `doctor`. The following info might help. A backquote[1] inhibits evaluation of all elements in a list, except for those preceded by a comma. E. g.

```
>(SETQ variable 'surprise)
SURPRISE
>`(this is a ,variable)
(THIS IS A SURPRISE)
```

If a variable in such a list is preceded by `,@` instead, it will be spliced into the result. E.g.

```
>(SETQ variable '(big surprise))
(BIG SURPRISE)
>`(this is a ,variable)
(THIS IS A (BIG SURPRISE))
>`(this is a ,@variable)
(THIS IS A BIG SURPRISE)
```

`DO` iterates until the `RETURN` form gets evaluated. In each iteration, local variables `sentence`, `a-list` and `mother` are temporarilly bound to the values returned by `READ`, `NIL` and `NIL` respectively. After that, the `COND` gets evaluated and the corresponding "psychiatrist's" messages are displayed. Then, variables `sentence`, `a-list`, `mother` get bound to the values returned by the last forms in their lines.

Exercise: Extend the list of restricted words. Add patterns to handle some more replies from the pacient.

[1] on your keyboard, it is situated to the left of the key 1