CSE 341 – PROGRAMMING LANGUAGES
MIDTERM REPORT

ELİF GORAL
171044003

I create a struct for fact. –> name(FirstEntry, SecondEntry):

Types are variable,objectName and number.

```
(defstruct fact
    name
    firstEntry
    firstEntryType
    secondEntry
    secondEntryType
)
```

I create a struct for rule.

```
(defstruct rule
    headOfRule
    bodyOfRule
)
```

That function checks the number is digit or not.

```
(defun isDigit (charCode)
      (if(and (< charCode 58) (> charCode 47))
          (return-from isDigit 1)
          (return-from isDigit 0)
      )
)
```

The list of parameters can have string or numeric entries. String entries starting with capital letters indicate that the parameter is a variable. String entries starting with lowercase letters indicate name of objects (NOTE: object names cannot start with caps). Numeric entries are treated as Common Lisp integers.

```
(defun checkEntry(input)
    ;   That condition checks the input is number or not.
    ;   If the input is number, return "number".
    (when(equal 1 (isDigit (char-code(aref input 0))))
        (return-from checkEntry "number")
    )

    ;   That condition checks the input is string or not.
    (when(stringp input)
        ;   That condition checks the string's first character is upper case or not.
        ;   If input is string and its first character is upper case, it means parameter is "variable"
        ;   If input is string and its first character is lower case, it means parameter is "name of the object".
        (if(upper-case-p (aref input 0))
            (return-from checkEntry "variable")
            (return-from checkEntry "objectName")
        )
    )
)
```

IsFact: Parameter fromRule's meaning : If fromRule 1 that means call from isRule function and that means end of the input has no ".". Control added for this. That function checks the input is fact or not. If the input has not :- and has "(, that means input is fact. After understanding the input is fact, check the fact's parameter size. If the fact has ",", that means fact has 2 parameter. Otherwise fact has 1 parameter. If fromRule parameter is one, I add the fact to the factsOfRules list. Otherwise I add fact to factList.

```
(defun isFact(input fromRule)
    (when(stringp input)
        (when(equal (search ":-" input) nil)
            (when(not (equal (search "(" input) nil))
                (setq bufferStruct (make-fact
                    :name nil
                    :firstEntry nil
                    :firstEntryType nil
                    :secondEntry nil
                    :secondEntryType nil
                    )
                )
                (setq name_fact (subseq input 0 (search "(" input)))
                (setf (fact-name bufferStruct) name_fact)
                (setq startIndex (search "(" input))
                (setq firstElementEndIndex (search "," input))
                (setq endIndex (search ")" input))
            )
            (when(equal firstElementEndIndex nil)
                (setq element (subseq input (+ 1 startIndex) endIndex))
                (setf (fact-firstEntry bufferStruct) element)
                (setf (fact-firstEntryType bufferStruct) (checkEntry element))
                (if(equal fromRule 1)
                    (push bufferStruct factOfRule)
                    (push bufferStruct factList)
                )
                (return-from isFact 1)
            )
            (when(not (equal firstElementEndIndex nil))
                (setq firstElement (subseq input (+ 1 startIndex) firstElementEndIndex))
                (if(equal 1 fromRule)
                    (setq secondElement (subseq input (+ 1 firstElementEndIndex) (- (length input) 1)))
                    (setq secondElement (subseq input (+ 1 firstElementEndIndex) (- (length input) 2)))
                )
                (setf (fact-firstEntry bufferStruct) firstElement)
                (setf (fact-secondEntry bufferStruct) secondElement)
                (setf (fact-firstEntryType bufferStruct) (checkEntry firstElement))
                (setf (fact-secondEntryType bufferStruct) (checkEntry secondElement))
                (if(equal fromRule 1)
                    (push bufferStruct factOfRule)
                    (push bufferStruct factList)
                )
                (return-from isFact 2)
            )
        )
    )
    (return-from isFact 0)
)
```

FindFactsOfRules: That function finds the facts of rules and send this facts to the isFact function.

```
(defun findFactsOfRules(input)
    (setq buffinput input)
    (loop
        (setq startIndex 0)
        (setq endIndex (search ")" buffinput))
        (setq element (subseq buffinput startIndex (+ 1 endIndex)))
        (isFact element 1 )
        (setq buffinput (subseq buffinput (+ 3 endIndex)))
        (setq endIndex (search ")" buffinput))
        (when(string= "." (aref buffinput (+ endIndex 1)))
            (setq element (subseq buffinput 0 (+ 1 endIndex)))
            (isFact element 1 )
            (return 1)
        )
    )
)
```

isRule(): That function checks the statement is horn clause or prediction. If input has ":-" that means it is horn clause statement. If the input is rule, add to ruleList.

```
(defun isRule(input)
    (when(stringp input)
        (when(not (equal (search ":-" input) nil))
            (setq bufferStructRule (make-rule
                    :headOfRule nil
                    :bodyOfRule nil
                )
            )
            (setq index (search ":-" input))
            (setq head (subseq input 0 index))
            (findFactsOfRules (subseq input (+ 3 index)))
            (setf (rule-headOfRule bufferStructRule) (subseq input 0 index))
            (setf (rule-bodyOfRule bufferStructRule) factOfRule)
            (push bufferStructRule ruleList)
            (return-from isRule 1)
        )
    )
    (return-from isRule 0)
)
```

IsQuery(): If input is not rule and start with ?-, that means input is query.

```
(defun isQuery(input)
    (when(stringp input)
        (when(equal (search ":-" input) nil)
            (when(not (equal (search "?-" input) nil))
                (return-from isQuery 1)
            )
        )
    )
    (return-from isQuery 0)
)
```

generateListForFact(): That function prints the identical fact of input.

```
(defun generateListForFact(input fromRule)
    (setq name (subseq input 0 (search "(" input)))
    (when(not(equal (search "," input) nil))
        (setq par1 (subseq input (+ 1 (search "(" input)) (search "," input)))
        (setq par2 (subseq input (+ 1 (search "," input)) (search ")" input)))
    )
    (when(equal (search "," input) nil)
        (setq par1 (subseq input (+ 1 (search "(" input)) (search ")" input)))
        (setq par2 nil)
    )
    (setq generateFactList nil)
    (isFact input 0)
    (defvar generatedList nil)

    (when(not (equal factList nil))
        (dolist (buffer factList)
            (when(string= name (fact-name buffer))
                (when(and (string= par1 (fact-firstEntry buffer)) (string= par2 (fact-secondEntry buffer)))
                    ;   tamamını kapsayan "("
                    (push (code-char 40) generatedList)
                        ;   fact'i kapsan "("
                        (push (code-char 40) generatedList)
                            ; fact-name
                            (setq str (concatenate 'string (string (code-char 34)) (fact-name buffer) (string (code-char 34))))
                            (push str generatedList)
                            ; tek elemanlı fact demek oluyor.
                            (when(equal (fact-secondEntry buffer) nil)
                                ;   elemanı kapsayan "("
                                (push (code-char 40) generatedList)
                                (if(or (string= (fact-firstEntryType buffer) "variable") (string= (fact-firstEntryType buffer) "objectName"))
                                    (setq str (concatenate 'string (string (code-char 34)) (fact-firstEntry buffer) (string (code-char 34))))
                                    (setq str (fact-firstEntry buffer))
                                )
                                (push str generatedList)
                                ;   elemanı kapatan ")"
                                (push (code-char 41) generatedList)
                            )
```

```
                            ; 2 elemanlı fact demek oluyor.
                            (when(not (equal (fact-secondEntry buffer) nil))
                                ;   elemanları kapsayan "("
                                (push (code-char 40) generatedList)
                                (if(or (string= (fact-firstEntryType buffer) "variable") (string= (fact-firstEntryType buffer) "objectName"))
                                    (setq str (concatenate 'string (string (code-char 34)) (fact-firstEntry buffer) (string (code-char 34))))
                                    (setq str (fact-firstEntry buffer))
                                )
                                (push str generatedList)
                                (if(or (string= (fact-secondEntryType buffer) "variable") (string= (fact-secondEntryType buffer) "objectName"))
                                    (setq str (concatenate 'string (string (code-char 34)) (fact-secondEntry buffer) (string (code-char 34))))
                                    (setq str (fact-secondEntry buffer))
                                )
                                (push str generatedList)
                                ;   elemanları kapatan ")"
                                (push (code-char 41) generatedList)
                            )
                            ; fact'i kapatan ")"
                            (push (code-char 41) generatedList)

                        (when(equal fromRule 0)
                            ;   "()"
                            (push (code-char 40) generatedList)
                            (push (code-char 41) generatedList)
                        )

                    ; tamamen fact'i kapatan ")"
                    (push (code-char 41) generatedList)
                )
            )


        )
    )
    (dolist (buffer (reverse generatedList))
        (format t "~a " buffer)
    )
    (push (reverse generatedList) resultList)

    (format t "~%")
    (setq generatedList nil)
)
```

generatedListForQuery(): That function prints the identical query of input.

```
(defun generatedListForQuery(input)
    (when(equal (isQuery input) 1)
        (defvar generatedList nil)

        ;(
        (push (code-char 40) generatedList)

        ; ()
        (push (code-char 40) generatedList)
        (push (code-char 41) generatedList)

        ; (
        (push (code-char 40) generatedList)

        (setq name (subseq input 2 (search "(" input)))
        (when(not(equal (search "," input) nil))
            (setq par1 (subseq input (+ 1 (search "(" input)) (search "," input)))
            (setq par2 (subseq input (+ 1 (search "," input)) (search ")" input)))

            (setq str_name (concatenate 'string (string (code-char 34)) name (string (code-char 34))))

            (if(or (string= (checkEntry par1) "variable") (string= (checkEntry par1) "objectName"))
                (setq str1 (concatenate 'string (string (code-char 34)) par1 (string (code-char 34))))
                (setq str1 par1)
            )

            (if(or (string= (checkEntry par2) "variable") (string= (checkEntry par2) "objectName"))
                (setq str2 (concatenate 'string (string (code-char 34)) par2 (string (code-char 34))))
                (setq str2 par2)
            )
            (push str_name generatedList)
            (push (code-char 40) generatedList)
            (push str1 generatedList)
            (push str2 generatedList)
        )
```

```
        (when(equal (search "," input) nil)
            (setq par1 (subseq input (+ 1 (search "(" input)) (search ")" input)))
            (if(or (string= (checkEntry par1) "variable") (string= (checkEntry par1) "objectName"))
                (setq str1 (concatenate 'string (string (code-char 34)) par1 (string (code-char 34))))
                (setq str1 par1)
            )
            (setq str_name (concatenate 'string (string (code-char 34)) name (string (code-char 34))))
            (push str_name generatedList)
            (push (code-char 40) generatedList)
            (push str1 generatedList)
        )

        (push (code-char 41) generatedList)
        (push (code-char 41) generatedList)
        (push (code-char 41) generatedList)

        (push (reverse generatedList) resultList)
        (dolist (buffer (reverse generatedList))
            (format t "~a " buffer)
        )
        (format t "~%")
        (setq generatedList nil)
    )
)
```

isFactTrue(): In this function, I compare the fact that comes with the input while I traversing the factlist. If the first parameter of input's type is "variable", I check the fact with put variable which comes as paramater,in its place. If it is the same fact, it is true. If the second parameter of input's type is "variable", I check the fact with put variable which comes as paramater,in its place. If it is the same fact, it is true.

```
(defun isFactTrue(input variable)
    (setq inputName (fact-name input))
    (setq inputFirstEntry (fact-firstEntry input))
    (setq inputFirstEntryType (fact-firstEntryType input))
    (setq inputSecondEntry (fact-secondEntry input))
    (setq inputSecondEntryType (fact-secondEntryType input))
    (dolist(buffer factList)
        (setq bufferName (fact-name buffer))
        (setq bufferFirstEntry (fact-firstEntry buffer))
        (setq bufferFirstEntryType (fact-firstEntryType buffer))
        (setq bufferSecondEntry (fact-secondEntry buffer))
        (setq bufferSecondEntryType (fact-secondEntryType buffer))
        (when(string= inputFirstEntryType "variable")
            (when(and (string= inputName bufferName) (string= inputSecondEntry bufferSecondEntry))
                (when(string= variable bufferFirstEntry)
                    (return-from isFactTrue 1)
                )
            )
        )
        (when(string= inputSecondEntryType "variable")
            (when(and (string= inputName bufferName) (string= inputFirstEntry bufferFirstEntry))
                (when(string= variable bufferSecondEntry)
                    (return-from isFactTrue 1)
                )

            )
        )
    )
    (return-from isFactTrue 0)
)
```

isQueryTrue(): Firstly, I learn the information about query. I keep that values in name,par1,par2. ---> 'name(par1,par2)'. If input has ',' , That means query has 2 parameter. After that, I search the query in ruleList which I filled before. I search with input has ',' or not approach with same as before. If the (query's name and current rule's name are equal) and (query's first parameter and current rule's first parameter are equal) .That means I will search this rule's facts and I will looking for a second parameter. I change Flag = 1 and looking for variable is second parameter. If the (query's name and current rule's name are equal) and (query's second parameter and current rule's second parameter are equal).That means I will search this rule's facts and I will looking for a first parameter. I change Flag = 1 and looking for variable is first parameter. I send the every facts to the isFactTrue function to understand fact is true or not. I and the values are returned from isFactTrue function. Then I search the factList. Final result is my query's result.

```
(defun isQueryTrue(input)
    ;   I learn the information about query
    (setq name (subseq input 3 (search "(" input)))
    (when(not(equal (search "," input) nil))
        (setq par1 (subseq input (+ 1 (search "(" input)) (search "," input)))
        (setq par2 (subseq input (+ 1 (search "," input)) (search ")" input)))
    )
    (when(equal (search "," input) nil)
        (setq par1 (subseq input (+ 1 (search "(" input)) (search ")" input)))
        (setq par2 nil)
    )
    (format t "name: ~a~%" name)
    (format t "par1: ~a~%" par1)
    (format t "par2: ~a~%" par2)

    (defvar result nil)

    ;   part of the searching in ruleList.
    (when(not (equal ruleList nil))
        (dolist (buffer ruleList)
            (setq flag 0)
            ;   finding the rule is same query
            (setq ruleName (subseq (rule-headOfRule buffer) 0 (search "(" (rule-headOfRule buffer))) )
            ;   query has 2 parameters:
            (when(not(equal (search "," (rule-headOfRule buffer)) nil))
                (setq rulePar1 (subseq (rule-headOfRule buffer) (+ 1 (search "(" (rule-headOfRule buffer))) (search "," (rule-headOfRule buffer)) ))
                (setq rulePar2 (subseq (rule-headOfRule buffer) (+ 1 (search "," (rule-headOfRule buffer))) (search ")" (rule-headOfRule buffer)) ))
            )
            ;   query has 1 parameters:
            (when(equal (search "," (rule-headOfRule buffer)) nil)
                (setq rulePar1 (subseq (rule-headOfRule buffer) (+ 1 (search "("  (rule-headOfRule buffer))) (search ")"  (rule-headOfRule buffer)) ))
                (setq rulePar2 nil)
            )
            ;   If the (query's name and current rule's name are equal) and  (query's first parameter and current rule's first parameter are equal);
            ;   that means I will search this rule's facts and I will looking for a second parameter.
            ;   I change Flag = 1 and looking for variable is second parameter.
            (when(and (string= name ruleName) (string= par1 rulePar1))
                ;(format t "name and par1 are same~%")
                (setq flag 1)
                (setq variable par2)
            )
```

```
            ;   I change Flag = 1 and looking for variable is second parameter.
            ;   If the (query's name and current rule's name are equal) and  (query's second parameter and current rule's second parameter are equal);
            ;   that means I will search this rule's facts and I will looking for a first parameter.
            (when(and (equal name ruleName) (string= par2 rulePar2))
                ;(format t "name and par2 are same~%")
                (setq flag 1)
                (setq variable par1)
            )
            (when(equal flag 1)
                (dolist(buffer2 (rule-bodyOfRule buffer))
                    ;   I send the every facts to the isFactTrue function to understand fact is true or not.
                    (push  (isFactTrue buffer2 variable) result)
                )
            )
        )

    )
    (setq a nil)
    (setq counter 1)
    (dolist(bufferResult result)
        (when(not(equal bufferResult nil))
            (if(equal bufferResult 1)
                (format t "~a th  fact is true~%" counter)
                (format t "~a th  fact is false~%" counter)
            )
            (setq counter (+ 1 counter))
            (setq a (* a bufferResult))
        )
    )

    ;   part of the searching in factList.
    (dolist(buffer factList)
        (when(string= (fact-name buffer) name)
            (format t "(fact-firstEntry buffer): ~a~%" (fact-firstEntry buffer))
            (format t "par1: ~a~%" par1)
            (format t "(fact-secondEntry buffer):~a~%"(fact-secondEntry buffer))
            (format t "par2:~a~%" par2)

            (when(and (string= (fact-firstEntry buffer) par1) (string= (fact-secondEntry buffer) par2))
                (setq a 1)
            )
        )
    )

    (if(equal a 1)
        (format t "~%final result: true~%")
        (format t "~%final result: false~%")
    )
    (format t "~%")
)
```

readFromFile(): In this function, I read from the file and find the categories of lines(query,fact,rule). Then call the generateList functions according the category. At the end, I call the writeToFile function and write to the "output.txt".

```
(defun readFromFile (fileName)
    (let ((in (open fileName :if-does-not-exist nil)))
        (when in
            (loop for line = (read-line in nil)
                while line do
                    (when(equal 1 (isQuery line))
                        (format t "case query~%")
                        (generatedListForQuery line)
                        (format t "----------------------------~%")
                        (isQueryTrue line)
                    )

                    (unless(equal 1 (isQuery line))
                        (when(equal 1 (isRule line))
                            (format t "case rule~%")
                            (generatedListForRule line)
                            (format t "----------------------------~%")
                        )
                        (unless(equal 1 (isRule line))
                            (format t "case fact~%")
                            (generateListForFact line 0)
                            (format t "----------------------------~%")
                        )
                    )
            )
            (printRule)
            (format t "printFacts:~%")
            (printFacts factList)
            (format t "~%")
            (format t "~%")
            (writeToFile "output.txt")
            (close in)
        )
    )
)
```

writeToFile():

```
(defun writeToFile(fileName)
    (with-open-file (stream fileName :direction :output)
        (dolist(buffer (reverse resultList))
            (format stream "~a~%" buffer)
        )
        (close stream)
    )
)
```

Test cases:

```
legs(X,2) :- mammal(X), arms(X,2).
legs(X,4) :- mammal(X), arms(X,0).
elif(Y,5) :- abc(2,Y), def(Y).
mammal(horse).
arms(horse,0).
arms(horse).
?- arms(horse,0).
```

Terminal results:

I didn't just write the equivalents and the result in lisp language. In addition, I showed ruleList and factLists. I wanted to show them because I perform control operations by storing the values in ruleList and factList.

```
elif@DESKTOP-LBAEPCM:/mnt/c/Users/Elif/Desktop$ clisp midterm.lisp
case rule
( ( "legs" ( "X" 2) ) ( ( "mammal" ( "X" ) ) ( "arms" ( "X" 2 ) ) ) )
-------------------------
case rule
( ( "legs" ( "X" 4) ) ( ( "mammal" ( "X" ) ) ( "arms" ( "X" 0 ) ) ) )
-------------------------
case rule
( ( "elif" ( "Y" 5) ) ( ( "abc" ( 2 "Y" ) ) ( "def" ( "Y" ) ) ) )
-------------------------
case fact
( ( "mammal" ( "horse" ) ) ( ) )
-------------------------
case fact
( ( "arms" ( "horse" 0 ) ) ( ) )
-------------------------
case fact
( ( "arms" ( "horse" ) ) ( ) )
-------------------------
case query
( ( ) ( " arms" ( "horse" 0 ) ) )
-------------------------
```

```
-------------------------

final result: true

rules:
------------------------

headOfRule:
elif(Y,5)

bodyOfRule:
name: def  firstEntry: Y  firstEntryType: variable  secondEntry: NIL secondEntryType: NIL
name: abc  firstEntry: 2  firstEntryType: number  secondEntry: Y secondEntryType: variable

-------------------------


headOfRule:
legs(X,4)

bodyOfRule:
name: arms  firstEntry: X  firstEntryType: variable  secondEntry: 0 secondEntryType: number
name: mammal  firstEntry: X  firstEntryType: variable  secondEntry: NIL secondEntryType: NIL

-------------------------


headOfRule:
legs(X,2)

bodyOfRule:
name: arms  firstEntry: X  firstEntryType: variable  secondEntry: 2 secondEntryType: number
name: mammal  firstEntry: X  firstEntryType: variable  secondEntry: NIL secondEntryType: NIL

-------------------------
```

```
------------------------

printFacts:
name: arms  firstEntry: horse  firstEntryType: objectName  secondEntry: NIL secondEntryType: NIL
name: arms  firstEntry: horse  firstEntryType: objectName  secondEntry: 0 secondEntryType: number
name: mammal  firstEntry: horse  firstEntryType: objectName  secondEntry: NIL secondEntryType: NIL
```

Output.txt results:

```
((( ( "legs" ( "X" 2) )) (( ( "mammal" ( "X" ) ) ( "arms" ( "X" 2 ) ) ) )))
((( ( "legs" ( "X" 4) )) (( ( "mammal" ( "X" ) ) ( "arms" ( "X" 0 ) ) ) )))
((( ( "elif" ( "Y" 5) )) (( ( "abc" ( 2 "Y" ) ) ( "def" ( "Y" ) ) ) )))
(( ( "mammal" ( "horse" ) ) ( ) ))
(( ( "arms" ( "horse" 0 ) ) ( ) ))
(( ( "arms" ( "horse" ) ) ( ) ))
(( ( ) ( " arms" ( "horse" 0 ) ) ))
True
```

Test case 2:

Input.txt

```
legs(X,2) :- mammal(X), arms(X,2).
legs(X,4) :- mammal(X), arms(X,0).
elif(Y,5) :- abc(2,Y), def(Y).
mammal(horse).
arms(horse,0).
arms(horse).
?- arms(horse,2).
```

```
case rule
( ( "legs" ( "X" 2) ) ( ( "mammal" ( "X" ) ) ( "arms" ( "X" 2 ) ) ) )
-------------------------
case rule
( ( "legs" ( "X" 4) ) ( ( "mammal" ( "X" ) ) ( "arms" ( "X" 0 ) ) ) )
-------------------------
case rule
( ( "elif" ( "Y" 5) ) ( ( "abc" ( 2 "Y" ) ) ( "def" ( "Y" ) ) ) )
-------------------------
case fact
( ( "mammal" ( "horse" ) ) ( ) )
-------------------------
case fact
( ( "arms" ( "horse" 0 ) ) ( ) )
-------------------------
case fact
( ( "arms" ( "horse" ) ) ( ) )
-------------------------
case query
( ( ) ( " arms" ( "horse" 2 ) ) )
-------------------------

final result: false

rules:
-----------------------
```

```
rules:
------------------------

headOfRule:
elif(Y,5)

bodyOfRule:
name: def  firstEntry: Y  firstEntryType: variable  secondEntry: NIL secondEntryType: NIL
name: abc  firstEntry: 2  firstEntryType: number  secondEntry: Y secondEntryType: variable

------------------------


headOfRule:
legs(X,4)

bodyOfRule:
name: arms  firstEntry: X  firstEntryType: variable  secondEntry: 0 secondEntryType: number
name: mammal  firstEntry: X  firstEntryType: variable  secondEntry: NIL secondEntryType: NIL

------------------------


headOfRule:
legs(X,2)

bodyOfRule:
name: arms  firstEntry: X  firstEntryType: variable  secondEntry: 2 secondEntryType: number
name: mammal  firstEntry: X  firstEntryType: variable  secondEntry: NIL secondEntryType: NIL

------------------------

printFacts:
name: arms  firstEntry: horse  firstEntryType: objectName  secondEntry: NIL secondEntryType: NIL
name: arms  firstEntry: horse  firstEntryType: objectName  secondEntry: 0 secondEntryType: number
name: mammal  firstEntry: horse  firstEntryType: objectName  secondEntry: NIL secondEntryType: NIL
```

Output.txt

```
((( ( "legs" ( "X" 2) )) (( ( "mammal" ( "X" ) ) ( "arms" ( "X" 2 ) ) ) )))
((( ( "legs" ( "X" 4) )) (( ( "mammal" ( "X" ) ) ( "arms" ( "X" 0 ) ) ) )))
((( ( "elif" ( "Y" 5) )) (( ( "abc" ( 2 "Y" ) ) ( "def" ( "Y" ) ) ) )))
(( ( "mammal" ( "horse" ) ) ( ) ))
(( ( "arms" ( "horse" 0 ) ) ( ) ))
(( ( "arms" ( "horse" ) ) ( ) ))
(( ( ) ( " arms" ( "horse" 2 ) ) ))
False
```