CSC421: Assignment 2

Student: Jordan Yu, V00727036
Date: February 8, 2015
Instructor : George Tzantakis

Table of Contents

Table of Figures

Question 1

This section presents the syntax used for processing propositional logic statements. The syntax is presented using BNF form in the following figure.

```
EXP => ATOM | COMPLEX
ATOM => True | False | IDEN
IDEN => [_a-zA-Z0-9]
COMPLEX => (EXP)
   |  ~EXP        // negation
   |  EXP * EXP   // logical AND
   |  EXP + EXP   // logical OR
   |  EXP -> EXP  // logical implication
   |  EXP / EXP   // logical if and only if

OPERATOR PRECEDENCE (lowest to highest): ~,*,+,->,/
```

FIGURE 1 : BNF FOR PROPOSITIONAL LOGIC

Each statement can be made up of either an atomic sentence or a complex sentence. Atomic sentences can be either True, False or a placeholder identifier. Identifiers are specified with ASCII characters. Complex sentence are sentences which apply an operator or is an expression surrounded by brackets. Binary operators are applied as "infix". The base syntax use the textbook's BNF form [1].

Question 2

A truth evaluator for propositional logic was implemented for this section. Python was used to implement the evaluator. The relevant source files for this section can be found in the code listings under Logic:

```
main_logic.py
logic.py
test_logic.py
```

The main program can be run with the command and opens up a REPL prompt:

```
python main_logic.py
```

The user enters expressions into the loop in the form.

IDEN = EXP

Example Usage:

```
 A = (p1 + p2) * p3
 P1 = True
 P2 = False
 P3 = True
 eval
> formula A : (p1 + p2) * p2 => True
exit
```


The expression is stored as a part of the state of the program with the given identifier. Four commands are also available to the user in order to evaluate the expressions.

Command	Description
eval	Evaluate all the expressions currently recorded in the program.
ls	Print out all the identifiers and expressions currently saved in the program.
clear	Clear all recorded/saved expressions
exit	Exit the REPL

TABLE 1: COMMANDS FOR MAIN_LOGIC.PY

The implementation of the evaluator and parser can be found in *logic.py.*
The logic.py file provides the function *eval_expr(expression,dict).*
The evaluator function takes two parameters; a string representation of the expression in the BNF form specified in Question 1, and a dictionary which maps currently known identifiers to expressions.

The inner workings of the evaluator works as follows.
The user passes in a string representing the expression to evaluate. The program parses the string into a list of tokens. Each token represents either an ATOM, IDEN, brackets, or one of the available operators. These tokens are processed and used to form an expression tree. This tree is then evaluated recursively on each node until the final result is determined.

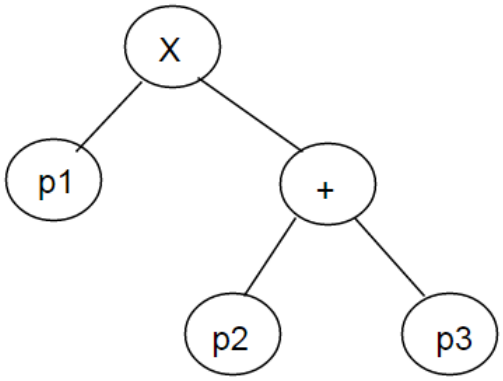

FIGURE 2 : EXPRESSION TREE FOR P1 * (P2 + P3)

For example evaluating the expression tree in figure 1
Given that we know:

p1 = True p2 = False p3 = p4 + P5
p4 = True p5 = False

The full expression of the represented by the tree is therefore
p1 * (p2 + (p4 + p5))
True * (False + (True + False))
True *(False + True)
True * True
True

The following formulas were evaluated with the propositional logic evaluator and the formula E was evaluated under two cases.

$A = \big((p_1 \rightarrow (p_2 \wedge p_3)) \wedge ((\neg p_1) \rightarrow (p_3 \wedge p_4))\big)$
$B = ((p_3 \rightarrow (\neg p_6)) \wedge ((\neg p_3) \rightarrow (p_4 \rightarrow p_1)))$
$C = ((\neg(p_2 \wedge p_5)) \wedge (p2 \rightarrow p_5))$
$D = (\neg(p_3 \rightarrow p_6))$
$E = ((A \wedge (B \wedge C)) \rightarrow D)$

Case 1 :
$p_2 = p_4 = p_6 = True$
$p_1 = p_3 = p_5 = False$
Result : E => True

Case 2 :
$p_2 = p_4 = p_6 = False$
$p_1 = p_3 = p_5 = True$
Result: E => True

The full input and output of the program can be found by running the program with the *input_logic* file (found in the code-listings).

Question 3

This section converts the following English sentences to FOL.

1. Every cruise ship was accompanied by at least one tug
$$\forall x, \exists y \quad Cruise(x) \rightarrow Accompany(x, y) \wedge Tug(y)$$

2. At least one tanker was accompanied by more than one tug
$$\exists x, y, w \quad Tanker(x) \wedge Tug(y) \wedge Tug(w) \wedge Accompany(x, y) \wedge Accompany(x, w) \wedge w \neq y$$

3. All the fishing boats but one returned safely to port.
$$\exists x \quad FishingBoat(x) \wedge \neg Safe(x)$$
$$\forall y \; FishingBoat(y) \wedge \neg Safe(y) \rightarrow x = y$$

4. There are exactly two students with grade less than B
$$\exists x, y \; Student(x) \wedge Student(y) \wedge GradeLessThan(x, B) \wedge GradeLessThan(y, B) \wedge x \neq y$$
$$\forall z \; Student(z) \wedge GradeLessThan(z, B) \rightarrow z = x \vee z = y$$

Question 4

This section describes the syntax and program for matching two formulas.
The following figure describes the BNF of the matching syntax.

```
EXP => FORMULA | IDEN | VAR
FORMULA => IDEN(EXP,...,EXP)
IDEN => [A-Z][_a-zA-Z0-9]     // normal ascii word to represent a value
VAR => [a-z]IDEN              // lower case identifier
```

FIGURE 3: BNF FOR MATCH

An expression is formed as either a formula, identifier or a variable. Formulas are expression which have an identifier and wrap a command separate tuple of expressions. Identifiers are capitalized ASCII words. Variables are identifiers which start with a lower case letter.

The relevant sources for the implementation can be found in the code listing section. The files of note include:

```
main_match.py
match.py
test_match.py
```

The program is run using

```
python main_match.py
```

This starts a REPL loop which evaluates user commands and evaluates matches for formulas.

Command	Description
match	Put the program into a state in which to receive two formulas to match. Each formula is entered onto their own lines. Once both formulas are entered, then the formulas are matched together and the result is printed out to the user.
exit	Exit the REPL

TABLE 2: COMMANDS FOR MAIN_MATCH.PY

Example Usage:

```
 python main_match.py
 match
 Brother(Fred,Son(John))
 Brother(x,Son(y))
> is valid for { x: Fred,  y: John}
 exit
```

The implementation of the matcher can be found in *match.py.*
The match.py file provides the function *match(expression1, expression2).*
The function takes two strings as represented in the BNF specified. The first expression is an expression specified using only formulas and identifiers. The second expression is an expression built using formulas and identifiers.

The inner workings of the evaluator works as follows.
The user passes in the strings to be matched. The program parses the string into an n-tree of sub-expressions. Each node of the tree represents either an identifier, formula or a variable. This expression

tree is then traversed for matches. As the program traverses through each node it records variable-to-identifier/formula assignments and checks for cases of conflicts. These cases include:

1. A new variable – No conflict, add to the dictionary recording the assignments.
2. Variable already recorded – Recursively match the recorded assignment with the candidate expression. If the match is positive then continue, else return that the entire expression fails to match.
3. Both nodes contain sub-formulas – Recursively match all the children of both formulas. If one of the children does not match then the entire expression does not match.

Once the entire tree is traversed, and no conflicts were found, the dictionary of variable-to-identifier/formula assignments is returned to the user.

Question 5

This section describes the prolog database of rules and facts which allows the user to create sub-lists which contains any consecutive duplicates of elements [2].

```
get_segment([],A,[],[]).
get_segment([X|Rest],A,[A|Seg],Result):-
   X = A,
   get_segment(Rest,A,Seg,Result).
get_segment([X|Rest],A,[],[X|Rest]):- X \= A.

make_sub([],[]).
make_sub([X|Rest],[Block|Result]):-
   get_segment([X|Rest],X,Block,Remain),
   make_sub(Remain,Result).
```

FIGURE 4: PROLOG RULES FOR MAKE_SUB

The prolog database contains two main rules *get_segment* and *make_sub*

get_segment (InputList,TargetElement, Rt_Segment, Rt_Remain)
The purpose of this rule is to extract out the consecutive elements in the provided list.
The consecutive elements from the front of the list are determined by Rt_Segment, and Rt_Remain denotes the remaining elements of the list.

make_sub(InputList,Rt)
This is the rule which allows the user to create sub-lists of consecutive elements. The rule is determined by two conditions. The first condition uses get_segment to extract consecutive elements from the list, and a recursive call to *make_sub* is used to process the remainder of the list.

Code Listing

This section contains all the source files created for this project. A full listing

Logic

```
# main_logic.py

import sys
import logic as logic


# dictionary holding the assignment of identifiers to expressions
d = {}

while(True):
  line = sys.stdin.readline()

  # empty line
  if(len(line) == 0):
    continue

  # remove the end-line character
  if(line[-1] == "\n"):
    line = line[:-1]

  # echo the command back to the user
  print(line)

  # split on the tokens to decide if it is a command
  # or an assignment
  toks = line.split("=")
  if( len(toks) == 1):
    # command
    if(toks[0] == "eval"):
      for e in d:
        if( d[e] == True or d[e] == False):
          continue
        print("formula {} : {} => {}".format(e,d[e],logic.eval_expr(d[e],d)) )

    elif(toks[0] == "ls"):
      for e in d:
        print("{} : {}".format(e,d[e]))

    elif(toks[0] == "clear"):
      d = {}
    elif(toks[0] == "exit"):
      exit()

  elif( len(toks) == 2):
    # an assignment
    iden = toks[0].strip()
    exp = toks[1].strip()
```

```
    if exp.lower() == "true":
      d[iden] = True
    elif exp.lower() == "false":
      d[iden] = False
    else:
      d[iden] = exp   # logic.py

operators = {
  "*" : 0, # and
  "+" : 0, # or
  "->" : 1, # implies
  "/" : 1, # iff
  "~" : 2, # negation
}
operator_args = {
  "*" : 2,
  "+" : 2,
  "->" : 2,
  "/" : 2,
  "~" : 1,
}
operator_fns = {
  "*" : lambda p : p[0] and p[1],
  "+" : lambda p : p[0] or p[1],
  "->" : lambda p : (not p[0]) or p[1],
  "/" : lambda p : (not p[0] and p[1]) and (not p[1] and p[0]),
  "~" : lambda p : not p[0],
}

class node:
  def __init__(self):
    # 0 means operator
    # 1 means value
    self.is_operator = False
    self.value = None
    self.children = []

  def set(self,elem):
    self.is_operator = elem in operators.keys()
    self.value = elem
    return self

  def __str__(self):
    if( self.is_operator):
      return "{{{} {}}}".format(self.value,self.children)
    else:
      return "{}".format(self.value)
```

```
  def __repr__(self):
    return self.__str__()

  def toLowestForm(self):
    if( self.is_operator):
      if(self.value == "->"):
        # convert to the form (~a + b)
        a = self.children[0]
        b = self.children[1]
        self.children = []

        # change to an "or" operator
        self.value = "+"

        # ~a
        left = node().set("~")
        left.children.append(a)
        self.children.append(left)

        # b
        self.children.append(b)

      elif(self.value == "/"):
        # convert into the form (a->b) * (b->a)
        a = self.children[0]
        b = self.children[1]

        # change to the "and" operator
        self.value = "*"

        # (a->b)
        p1 = node().set("->")
        p1.children = [a,b]
        p1.toLowestForm()

        # ( b->a)
        p2 = node().set("->")
        p2.children = [b,a]
        p2.toLowestForm()

        self.children = []
        self.children.append(p1)
        self.children.append(p2)

    return self

"""
```

```
@purpose - process the string expression into a list of tokens
@parameter e - the string representing the expressoin
  (i.e  a * b + (c -> d) )
@return - return the list tokens
  (i.e [ "a","*","b","+","(","c","->","d",")" ])
"""
def processIntoTokens(e):
  e = "".join(e.split())
  tokens = []
  w = ""
  skip_flag = False
  left_count = 0
  right_count = 0

  for i in range(0,len(e)):
    # see if we should skip the the next token
    if(skip_flag):
      skip_flag = False
      continue

    c = e[i]

    # if the token is an operator then just append it
    if( c in ["(",")","*","+","~","/"]):

      # keep counts on the number of brackets
      if( c == "("):
        left_count += 1
      if( c == ")"):
        right_count += 1

      # if we were recording a identifier, dump it
      # into the tokens array
      if(len(w)!= 0):
        tokens.append(w)
        w = ""

      # add the operator to the token list
      tokens.append(c)

    elif(c=="-"):
      # special case for the '->' operator
      if( i+1 < len(e) and e[i+1]=='>'):

        #record the identifier
        if(len(w)!= 0):
          tokens.append(w)
          w = ""
```

```
          # add the operator
          tokens.append("->")
          # we want to skip the token
          skip_flag = True
        else:
          raise Exception("Invalid Expression")
      else:
        # record the next character in the identifier
        w += c

  # dump the last identifier
  if(len(w) != 0):
    tokens.append(w)

  if(left_count != right_count):
    # ensures matching brackets
    raise Exception("Invalid Expression")
  else:
    return tokens


"""
@purpose: determine the index of the matching brackets
@parameter e: the token list in which to search for the matching brackets
@parameter i: the current position index
@return : the index of the matching bracket
"""
def posMatchingBracket(e,i):
  left_count = 0
  for j in range(i,len(e)):
    c = e[j]
    if( c == "("):
      left_count += 1
    elif( c == ")"):
      left_count -=1
      if(left_count == 0):
        return j

"""
@purpose - recursively parse the tokens list an creates an expression tree.
@param e - a list of tokens. Run the expression throughh processIntoTokens
      in order to create the list of tokens
@return - return a node object.
@references - http://www.engr.mun.ca/~theo/Misc/exp_parsing.htm
"""
def parseExpressionTokens(e):
  i = 0
```

```
size = len(e)

# helper funtion which creates an operator node.
# it takes the required number of arguments from the elem list
def apply(op,elems):
  num_args = operator_args[op.value]

  # check to see that we have enough arguments
  if(len(elems) < num_args):
    raise Exception("Invalid Expression")

  # create the node with the operator and the required arguments
  op.children = []
  for j in range(0,num_args):
    op.children = [elems.pop()] + op.children

  op = op.toLowestForm()
  return op

ops = []
elems = []
while( i < size):
  c = e[i]
  if c == "(" :
    # we have a sub-expression. evaulate this recursively
    # and place it onto the element stack
    end = posMatchingBracket(e,i)
    rs = parseExpressionTokens(e[i+1:end])
    elems.append(rs)
    i = end

  elif c in operators.keys():
    # we are processing an operator

    if(len(ops) > 0):
      # there are some operators to compare against
      cand = operators[c]
      champ = operators[ops[-1].value]

      # if we are trying to push an operator with lower precedence
      # on top of an operator with greater precendence
      # we pop off the operator and create a node for that operator
      # (popping off the required number of elements
      #    from the element stack)
      if( cand <= champ):
        op = ops.pop()
        op = apply(op,elems)
```

```
            # push the result back onto the element stack
            elems.append(op)

        # add the operator
        n = node().set(c)
        ops.append(n)
      else:
        n = node().set(c)
        elems.append(n)
      i += 1

    # process the rest of the operators with the remaining arguemnts
    while(len(ops) > 0 ):
      op = apply(ops.pop(),elems)
      elems.append(op)

    return elems[0]


"""
@purpose - evaluate the expression tree given by e, using the predicate
  specifed in the dictionary
@parameter e - is a node that represents the root of the expression tree in
  which we want to evaluate
@parameter d - a dictionary of assignments to the variables
"""
def evalExpressionTree(e,d):
  def _eval(e):
    if e.is_operator:
      children_rs = tuple(map(_eval,e.children))
      return operator_fns[e.value](children_rs)
    else:
      if e.value.lower() in "true":
        return True
      elif e.value.lower() == "false":
        return False
      elif e.value in d:
        rs = d[e.value]
        if( rs == True or rs == False):
          return rs
        else:
          return eval_expr(rs,d)
      else:
        raise Exception("\'{}\' identifier not specified".format(e.value))
  return _eval(e)


"""
```

```
@purpose - convenience function which
     processes the expression into a token list
     parses the token list into an expression tree
"""
def parse_expr(e):
  toks = processIntoTokens(e)
  return parseExpressionTokens(toks)

"""
@purpose - convenience function which
     processes the expression into a token list
     parses the token list into an expression tree
     evaluates the expression tree into the given value
"""
def eval_expr(e,d):
  toks = processIntoTokens(e)
  return evalExpressionTree(parseExpressionTokens(toks),d)# test_logic.py

import logic as logic

"""
@purpose - Test the logic expression evaluator.
  Failed tests will print to the screen with the test number and
     the failed expression
  Passed tests will NOT print anything.
"""

d = {
  "a" : True,
  "b" : False,
  "c" : True,
  "d" : True,
  "e" : True,
  "f" : True,
  "doodoo" : True,
  "booboo" : False
}

global_fail_flag = False
fail_count = [0]
num_tests = [0]

def attempt(num,expected,e):
  rs = False
  try:
    num_tests[0]+=1
    rs = (expected == logic.eval_expr(e,d))
  finally:
```

```
    if( rs != True):
      global_fail_flag = True
      fail_count[0]+= 1
      print("{:<10d} : {}".format(num,e) )

attempt(1,False,"a * b");
attempt(2,True ,"a + b");
attempt(3,False,"a -> b");
attempt(4,False,"a / b");
attempt(5,False,"~a");
attempt(6,True,"(a * b) + doodoo + booboo");
attempt(7,True,"((a * b) + doodoo + booboo)");
attempt(8,True,"((a * b) + (c * d)) -> f");

print("{}/{} Tests Passed".format(num_tests[0] - fail_count[0],num_tests[0]))
```

Match

```
import sys
import match

state = 2
exps = ["",""]

while(True):
  line = sys.stdin.readline()

  # empty line
  if(len(line) == 0):
    continue

  # remove the end-line character
  if(line[-1] == "\n"):
    line = line[:-1]

  # if the state is 2 then we are looking for input from the user
  if( state == 2):
    if( line == "match"):
      state = 0
      exps = []
    elif( line == "exit"):
      exit()
  else:
    # look for two lines of expression
    exps.append(line)
    state = (state + 1)%3

    if( state == 2):
```

```
        rs,d = match.match(exps[0],exps[1])
        if( rs == True):
          print("is true with " + str(d))
        else:
          print("fails")
class Node:
  def __init__(self):
    self.isVariable = False

    # for constants - just the string (i.e George)
    # for variables - the identifier (i.e x)
    # for formulas - The relation name ( i.e BrotherOf )
    self.value = None

    # if the exp is a formula then this will contain a list
    # of all the children nodes
    # i.e
    # (Brother(George,Dog(Fred)))
    # [George,Dog(Fred)]
    self.children = []

  """
  @purpose - convert the node into its string representation
    This is important to keep the given format as it is necesary
    for the test cases
  """
  def __str__(self):
    if( self.isVariable):
      return str(self.value)
    elif(len(self.children) == 0):
      return str(self.value)
    else:
      size = len(self.children)
      s = self.value + "("
      for i in range(0,size):
        s += str(self.children[i])
        if( i != size-1):
          s += ","
      s += ")"
      return s

  def __repr__(self):
    return self.__str__()

"""
@purpose - helper function used to determine if the given expression
  is a variable or if it is a constant/formula
@parameter e - (string) the expression to evaluate
```

```
"""
def isVariable(e):
  if len(e.split("(")) != 1:
    return False
  if( e[0].isupper()):
    return False

  return True

"""
@purpose - parse the given string expression and form an expression tree.
@parameter e - (string) the expression in which to parse.
@return - a Node which represents the root node of the expression tree
"""
def parse(e):
  # remove all the whitespace from the string
  e = "".join(e.split())

  # Create a new node object
  exp = Node()
  exp.isVariable = isVariable(e)

  if( exp.isVariable == True):
    # the expression is just a constant or an expression
    exp.value = e
    exp.children = []
    return exp

  # This is a formula or constant.
  # retrieve the name of the formula or constant
  exp.value = ""
  left_count = 0
  i = 0
  w = ""
  while( i < len(e)):
    c = e[i]
    i += 1
    # if the expression is a formula, the name ends
    # when we read the first '('
    if( c == "("):
      left_count += 1
      break
    w += c
  exp.value = w

  # parse the rest of the formula for the arguments
  w = ""
  while( i < len(e)):
```

```
    c = e[i]
    i += 1

    if c == "(":
      left_count += 1
    elif  c == ")":
      left_count -= 1
      if( left_count == 0):
        # we have reached the end of the formula
        break
    elif c == ",":
      if(left_count == 1):
        # dump the recorded word
        # recusively call parse on the word
        exp.children.append(parse(w))
        w = ""
        continue
    w += c

  # make sure that the brackets are balanced
  if( left_count != 0):
    raise Exception("Invalid Expression. Unmatched brackets.")

  # add the last word into the list
  if len(w) != 0:
    exp.children.append(parse(w))

  return exp

"""
@purpose - determine if exp1 matches exp2
@parameter exp1 - (string) expression in which to match.
  Cannot contain any vairables
@parameter exp2 - (string) expression which contains
  formulas, constants or variables
@return (bool,dict) - returns a pair which represents the outcome of the match
  if true, then dict contains the dictionary of assignments
  if false, then dict is {}
"""
def match(exp1,exp2):

  # e1 - expression tree which only contains formulas or constants
  # e2 - expression tree which contains formulas, constants or variables
  # d  - a dictionary holding the currently recorded assignments
  #  Note, that d is modified mutated through recursive calls to the _match
  # return - return true if the expressions match, false otherwise
  def _match(e1,e2,d):
    if( e2.isVariable):
```

```
        # e2 is already assigned in the dict d
        if(e2.value in d):
          #variable is already assigned in the dictionary
          if _match(e1,d[e2.value],d):
            # if e1 matches with the dictioanry then we are okay.
            return True
          else:
            # e1 does not match with the dictionary so false
            return False
        else:
          # this is a new variable, assign it and keep going
          d[e2.value] = e1
          return True
      else:
        # e1 cannot be a variables
        if e1.isVariable:
          raise Exception("Invalid Expression")

        # e1 and e2 must have the same value
        if(e1.value != e2.value):
          return False

        # children length must match
        if(len(e1.children) != len(e2.children)):
          return False

        # all the children must match
        for i in range(0,len(e1.children)):
          rs = _match(e1.children[i],e2.children[i],d)
          if( rs == False):
            return False

        return True

    d = {}
    rs = _match(parse(exp1),parse(exp2),d)
    if( rs == True):
      d = { k : str(v) for k,v in d.items()}
    else:
      d = {}
    return (rs,d)
import sys
import match

global_fail_flag = False
fail_count = [0]
num_tests = [0]
```

```
def attempt(num,expected,e1,e2):
  rs = False
  try:
    num_tests[0]+=1
    rs = (expected == match.match(e1,e2))
  finally:
    if( rs != True):
      global_fail_flag = True
      fail_count[0]+= 1
      print("{:<10d} : {},{}".format(num,e1,e2) )

attempt(1,
  (True,{"x":"Fred","y":"George"}),
  "Brother(Fred,George)",
  "Brother(x,y)"
)

attempt(2,
  (True,{"x":"Dog(Fred)","y":"Dog(George)"}),
  "Brother(Dog(Fred),Dog(George))",
  "Brother(x,y)"
)

attempt(3,
  (True,{"x":"Fred","y":"Dog(George)"}),
  "Brother(Dog(Fred),Dog(George))",
  "Brother(Dog(x),y)"
)

attempt(4,
  (False,{}),
  "Brother(Dog(Fred),Dog(George))",
  "Brother(Dog(x),Dog(x))"
)

attempt(5,
  (True,{"x":"A"}),
  "Family(Mother(A),Father(A),A)",
  "Family(Mother(x),Father(x),x)"
)

attempt(6,
  (True,{"x":"A"}),
  "Family(Mother(A),Father(A),A)",
  "Family(Mother(x),Father(A),x)"
)

attempt(7,
```

```
  (True,{"x":"A","y":"A"}),
  "Family(Mother(A),Father(A),A)",
  "Family(Mother(x),Father(y),x)"
)

attempt(8,
  (False,{}),
  "Family(Mother(A),Father(A),A)",
  "Family(Mother(x),y,y)"
)

attempt(9,
  (True,{"x":"A","y":"Father(A)"}),
  "Family(Mother(A),Brother(Father(A)),Father(A))",
  "Family(Mother(x),Brother(y),y)"
)
attempt(10,
  (True,{"x":"A","y":"Father(A)"}),
  "Family(Mother(A),Father(A),Brother(Father(A)))",
  "Family(Mother(x),y,Brother(y))"
)

print("{}/{} Tests Passed".format(num_tests[0] - fail_count[0],num_tests[0]))
```

Input Files

```
// input_logic
A = (( p1 -> (p2 * p3)) * ((~p1) -> (p3 * p4)))
B = ((p3 -> (~p6)) * ((~p3) -> (p4 -> p1)))
C = ((~(p2 * p5)) * (p2 -> p5))
D = (~(p3 -> p6))
E = ((A * (B * C)) -> D)

p1 = false
p3 = false
p5 = false
p2 = true
p4 = true
p6 = true

eval

A = (( p1 -> (p2 * p3)) * ((~p1) -> (p3 * p4)))
B = ((p3 -> (~p6)) * ((~p3) -> (p4 -> p1)))
C = ((~(p2 * p5)) * (p2 -> p5))
D = (~(p3 -> p6))
E = ((A * (B * C)) -> D)
```

```
p1 = true
p3 = true
p5 = true
p2 = false
p4 = false
p6 = false
eval
exit
// input_match
match
Brother(Fred,George)
Brother(x,y)

match
Brother(Dog(Fred),George)
Brother(x,y)

match
Brother(Dog(Fred),Dog(George))
Brother(x,y)

match
Brother(Dog(Fred),Dog(George))
Brother(Dog(x),Dog(x))

match
Family(Mother(A),Father(A),A)
Family(Mother(x),Father(x),x)

match
Family(Mother(A),Father(A),A)
Family(Mother(x),Father(A),x)

match
Family(Mother(A),Father(A),A)
Family(Mother(x),Father(y),x)

match
Family(Mother(A),Father(A),A)
Family(Mother(x),y,y)

match
Family(Mother(A),Brother(Father(A)),Father(A))
Family(Mother(x),Brother(y),y)

match
Family(Mother(A),Father(A),Brother(Father(A)))
Family(Mother(x),y,Brother(y))
```

exit

References

[1] P. N. Stuart Russel, Artificial Intelligence : A Modern Approach Third Edition, Upper Saddle River, New Jersey: Prentice Hall, 2010.

[2] T.-J. T. Brna Paul, "Introduction to Prolo," 08 10 1996. [Online]. Available: http://www.doc.gold.ac.uk/~mas02gw/prolog_tutorial/prologpages/index.html#menu. [Accessed 07 02 2015].