CSC 421 Assignment 3

Cameron Long V00748439

Propositional Logic 1

The syntax is based around making the input to the system use the most basic of Boolean operators (and,or,not) in order to evaluate the sentences. Input to the system is also as close to English as possible. For example, the formula A would look like this:

"(not p or (q and r)) and (p or (r and s))"

The equivalencies are first removed with implications, then implications are substituted for OR and NOT logic.

Propositional Logic 2

The inputs to the system were from a text file on the Desktop of the local machine. The inputs to the evaluator produce the following results:

Input	*Result*
(not p1 or (p2 and p3)) and (p1 or (p3 and p4))	False
((not p3 or p6) and ((p3) or (not p4 or p1))	False
((not(p2 and p5) and not (p2 or p5))	False
p3 or not p6	False
(not(A and(B and C))or D)	True
p1 or not (p2 and p3)	True
p1 or p2 or p3	True
not(p1 and p2)	True
not not p1	False

English to First Order Logic

$$\exists x, y\ ProgrammingTeam(x) \cap Accompany(x, Professor) \cap \neg(x = y)$$

$$\forall x(programmingTeam(x) \cap Accompany(x, Professor) \cap (x = y)$$

$$\exists x, y(student(x) \cap Grade(x, z) \cap (\ Z > B), \cap Student(y) \cap Grade(y, g) \cap (g >' B')$$

$$\forall x(ProgrammingTeams(x) \cap ReturnHome(x, University)$$

Matching

This algorithm is a simple parser that breaks apart first order logic statements to a maximum of two nested statements. The statements are encoded similar to the assignment paper, but the outer function is met with square brackets for its arguments. The algorithm will compare two statements and the bindings to determine a match. It will also attempt to satisfy a determination for failed assignments, but its hit and miss. It responded to the test cases as such:

Input	*Result*
Brother(Fred) Brother(x) Louis	The statements are not equivalent: Valid if x= Fonzi
Brother(Fred) Brother(x) Fonzi	Statement is Equivalent
Dog(George) Dog(x) Louis	Statement is Equivalent
Dog(George) Dog(x) Theodore	The statements are not equivalent: Valid if x= Louis
Loves(Fred) Love(x) True	Statement is equivalent
Loves(George) Love(x) False	Statement is equivalent
Loves[Dog(Fred),Fred] Loves(x,y) Dog(Fred) Fred	Statement is equivalent
Loves[Dog(Fred),Fred] Loves(x,x) Dog(Fred) Dog(Fred)	Statements are not equivalent

The algorithm struggles to output correct assignments for multi-layered inputs. The arguments are split with spaces, so the program will evaluate each statement recursively and stitch it all together.

Code

Matching Part

```
import sys
def loves(dog, person):

    isLove = {'Fred': 'True', 'George': 'False'}

    actuallove = isLove[dog]

    if dog != actuallove:
        print("The statements are not equivalent: Valid if x= " +
str(actuallove))
        return False
    print("Statement is equivalent")
    return True


def doglove(person):

    dicts = {'Fred': 'Jerry', 'George': 'Sperry'}

    return dicts[person]


def hasdog(person, dog):
    dogName = {'Fred': 'Francis', 'George': 'Louis'}
```

```
    actualname = dogName[person]

    if actualname != dog:
        print("The statements are not equivalent: Valid if x= " + actualname)
        return False
    print("Statement is equivalent")
    return True

def dog(person):

    dogs = {'Fred': 'Jerry', 'George': 'Sperry'}

    return dogs[person]

def isbrother(person, brother):

    brothers = {'Fred': 'Fonzi', 'George': 'Theodore'}
    actualbrother = brothers[person]

    if brother != actualbrother:
        print("The statements are not equivalent: Valid if x= " +
actualbrother)
        return False
    print("Statement is Equivalent")
    return True


def main():

    file = open('C:\\Users\\Cameron\\Desktop\\part3test.txt', 'r')

    for line in file:
        line = line.rstrip('\n')
        splits = line.split(' ')
        dog1 = ""

        statement1 = splits[0]
        statement2 = splits[1]
        arg1 = statement1[statement1.index("(")+1:statement1.index(")")]
        arg2 = statement2[statement2.index("(")+1:statement2.index(")")]
        binding = splits[2]
        #print(statement1)
        compare1 = ""
        start =""
        end = ""
        if "[" in statement1:
            newfunction =
statement1[statement1.index("[")+1:statement1.index("]")]

            calls = newfunction.split(',')

            #print(calls)
            #print(newfunction)

            newstatement1 = calls[0]
            newstatement2 = calls[1]
            #print(newstatement1)
```

```
            #print(newstatement2)
            if "Dog" in newstatement1:
                dog1 = dog(newstatement2)
                #print(dog1)
            if "Loves" in statement1:
               # print(doglove(newstatement2))
                if doglove(newstatement2) == dog1:
                    start = "Fred loves " + dog1

            compare2 = splits[1]
            compare3 = splits[2]
            compare4 = splits[3]

            if "Dog" in compare3:
                dog2 =
dog(compare3[compare3.index("(")+1:compare3.index(")")])
            if "Loves" in compare2:
                if doglove(compare4) == dog2:
                    end = "Fred loves " + dog2

            if start == end:
                print("statements are equivalent")
            else:
                print("Statements are not equivalent")
            break
        if "Brother" in statement1:
            isbrother(arg1, binding)
        if "Dog" in statement1:
            hasdog(arg1, binding)
        if "Loves" in statement1:
            loves(arg1, binding)


if __name__ == "__main__":

    main()
```

Logic Part

```
from pyparsing import infixNotation, opAssoc, Keyword, Word, alphas

# define classes to be built at parse time, as each matching
# expression type is parsed
class BoolOperand(object):
    def __init__(self,t):
        self.label = t[0]
        self.value = eval(t[0])
    def __bool__(self):
        return self.value
    def __str__(self):
        return self.label
    __repr__ = __str__
    __nonzero__ = __bool__

class BoolBinOp(object):
    def __init__(self,t):
        self.args = t[0][0::2]
```

```
    def __str__(self):
        sep = " %s " % self.reprsymbol
        return "(" + sep.join(map(str,self.args)) + ")"
    def __bool__(self):
        return self.evalop(bool(a) for a in self.args)
    __nonzero__ = __bool__
    __repr__ = __str__

class BoolAnd(BoolBinOp):
    reprsymbol = '&'
    evalop = all

class BoolOr(BoolBinOp):
    reprsymbol = '|'
    evalop = any


class BoolNot(object):
    def __init__(self,t):
        self.arg = t[0][1]
    def __bool__(self):
        v = bool(self.arg)
        return not v
    def __str__(self):
        return "~" + str(self.arg)
    __repr__ = __str__
    __nonzero__ = __bool__

TRUE = Keyword("True")
FALSE = Keyword("False")
boolOperand = TRUE | FALSE | Word(alphas,max=1)
boolOperand.setParseAction(BoolOperand)

# define expression, based on expression operand and
# list of operations in precedence order
boolExpr = infixNotation( boolOperand,
    [
    ("not", 1, opAssoc.RIGHT, BoolNot),
    ("and", 2, opAssoc.LEFT,  BoolAnd),
    ("or",  2, opAssoc.LEFT,  BoolOr),
    ])


if __name__ == "__main__":
    p1 = False
    p2 = True
    p3 = False
    p4 = True
    p5 = False
    p6 = True
    q = True
    r = True
    s = True
    f = open('C:\\Users\\Cameron\\Desktop\\input.txt', 'r')

    print("p =", p1)
    print("q =", p2)
```

```
print("r =", p3)
print()

for line in f:
    t = line
    res = boolExpr.parseString(t)[0]
    success = "PASS" if bool(res) == expected else "FAIL"
    print (t,'\n', res, '=', bool(res),'\n', success, '\n')
```