Homework #2
Due Tu. 10/9

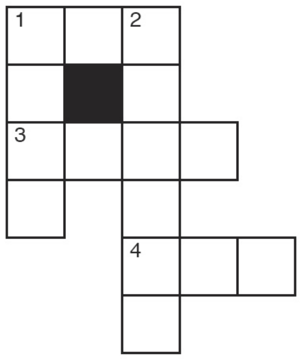
Please prepare a report including the source code, captures of output and send it along with related source files as a zip file to my blackboard. The zip file should be named by your family name (i.e., family-name.zip). Please print the report and bring the hard copy by the due date to the class. The selective grading may be used for homeworks of this course.

1. Suppose we have variables $\{A, B, C, D\}$ with domains of $\{1, 2, 3, 4\}$ for each one of the variables. Draw the constraint network and apply domain constraint and GAC algorithms to simplify the domain and find solutions for following constraint:
   $\{(A \neq 3), (B \neq 2), (C \neq 1), (A < B), (B \neq C), (C < D)\}$

2. Construct the truth table for following statements:
   (a) $(p \Rightarrow q) \wedge (\neg p \Leftrightarrow q)$
   (b) $(\neg p \Rightarrow (q \wedge \neg r))$

3. Refer to the Python codes downloaded for the first homework (`http://artint.info/AIPython/`). Refer to the `searchProblem.py` and `searchGeneric.py` and answer following questions.
   (a) Draw the graphs defined for problem 1 and problem 2.
   (b) Look at the comments below `searchGeneric.py` as guideline and perform DFS and A^* search on problems 1 and 2. Capture screen the outputs.
   (c) Explain how $AStarSearcher$ function can be modified/overridden to perform least-cost and greedy best searches.

4. Refer to the `cspExamples.py` and `cspSearch.py` and answer following questions.
   (a) Draw the graphs defined for csp1 and csp2.
   (b) Search methods can be used to solve the constraint satisfcation problems such as crossword puzzle. (See Figure 1). Refer to `cspSearch.py` and look at comments below the code for guideline. Solve the crossword1. Capture screen the outputs.



Words:
ant, big, bus, car, has, book, buys, hold, lane, year, ginger, search, symbol, syntax.

Figure 1: A crossword puzzle to be solved

5. Before we start logic programming in Python, we need to install a couple of packages. The package kanren is a Python package that enables logic programming in Python. We will also be using SymPy for some of the problems.

```
$ pip3 install kanren
$ pip3 install sympy
```

We encounter mathematical operations all the time. Logic programming is a very efficient way of comparing expressions and finding out unknown values. Let's see how to do that. Create a new Python file and import the following packages:

```
from kanren import run, var, fact
import kanren.assoccomm as la
```

Define a couple of mathematical operations:

```
# Define mathematical operations
add = 'addition'
mul = 'multiplication'
```

Both addition and multiplication are commutative operations. Let's specify that:

```
#Declare that these operations are commutative
#using the facts system
fact(la.commutative, mul)
fact(la.commutative, add)
fact(la.associative, mul)
fact(la.associative, add)
```

Let's define some variables:

```
# Define some variables
a, b, c = var(‘a‘), var(‘b‘), var(‘c‘)
```

Consider the following expression:
$expression\_orig = 3 \times (-2) + (1 + 2 \times 3) \times (-1)$

Let's generate this expression with masked variables. Consider three same equations:
$expression1 = (1 + 2 \times a) \times b + 3 \times c$
$expression2 = c \times 3 + b \times (2 \times a + 1)$
$expression3 = (((2 \times a) \times b) + b) + 3 \times c$

Our goal is to match these expressions with the original expression to extract the unknown values:

```
# Generate expressions
expression_orig = (add, (mul, 3, -2), (mul, (add, 1, (mul, 2, 3)), -1))
expression1 = (add, (mul, (add, 1, (mul, 2, a)), b), (mul, 3, c))
expression2 = (add, (mul, c, 3), (mul, b, (add, (mul, 2, a), 1)))
expression3 = (add, (add, (mul, (mul, 2, a), b), b), (mul, 3, c))
```

Compare the expressions with the original expression. The method run is commonly used in `kanren`. This method takes the input arguments and runs the expression. The first argument is the number of values, the second argument is a variable, and the third argument is a function:

```
# Compare expressions
print(run(0, (a, b, c), la.eq_assoccomm(expression1, expression_orig)))
print(run(0, (a, b, c), la.eq_assoccomm(expression2, expression_orig)))
print(run(0, (a, b, c), la.eq_assoccomm(expression3, expression_orig)))
```

Run the code, capture the screen and explain the output.

6. Let's see how to use logic programming to check for prime numbers. We will use the constructs available in `kanren` to determine which numbers in the given list are prime, as well as finding out if a given number is a prime or not. Create a new Python file and import the following packages:

```
from kanren import isvar, run, membero
from kanren.core import success, fail, goaleval, condeseq, eq, var
from sympy.ntheory.generate import prime, isprime
import itertools as it
```

Next, define a function that checks if the given number is prime depending on the type of data. If it's a number, then it's pretty straightforward. If it's a variable, then we have to run the sequential operation. To give a bit of background, the method conde is a goal constructor that provides logical AND and OR operations. The method `condeseq` is like `conde`, but it supports generic iterator of goals:

```
# Check if the elements of x are prime
def prime_check(x):
     if isvar(x):
        return condeseq([(eq, x, p)] for p in map(prime,it.count(1)))
     else:
        return success if isprime(x) else fail
```

Declare the variable x that will be used:

```
# Declate the variable
x = var()
```

Define a set of numbers and check which numbers are prime. The method `membero` checks if a given number is a member of the list of numbers specified in the input argument:

```
# Check if an element in the list is a prime number
list_nums = (23, 4, 27, 17, 13, 10, 21, 29, 3, 32, 11, 19)
print('\nList of primes in the list:')
print(set(lc.run(0, x, (membero, x, list_nums), (prime_check, x))))
```

Let's use the function in a slightly different way now by printing the first 7 prime numbers:

```
# Print first 7 prime numbers
print('\nList of first 7 prime numbers:')
print(run(7, x, prime_check(x)))
```

Run the code, and capture the output screen.