Spring, 2019 Name: _____

(Please *don't* write your id number!)

Exam 2: Python Programming with Recursion and Loops

Directions

The code for the `LispList` type used in the homework and in some problems on this exam is found in Figure 1 on the following page.
For this exam you are permitted one sheet (8.5 x 11 inches) of paper of notes on one side. It is a good idea to condense your notes into a small amount of ready reference material.
If you need more space, use the back of a page. Note when you do that on the front.
Before you begin, please take a moment to look over the entire test so that you can budget your time.
Clarity is important; if your answers are sloppy and hard to read, you may lose some points.

For Grading

Question:	1	2	3	Total
Points:	30	35	35	100
Score:				

```
# $Id: LispList.py,v 1.1 2017/02/27 04:52:50 leavens Exp $
import abc
class LispList(abc.ABC):
    pass
class Nil(LispList):
    def __init__(self):
        """Initialize this empty list"""
        pass
    def __eq__(self, lst):
        """Return True just when lst is also an instance of Nil."""
        return isinstance(lst, Nil)
    def __repr__(self):
        """Return a string representing this Nil instance."""
        return "Nil()"
    def __str__(self):
        """Return a string showing the elements of self."""
        return "[]"
    def isEmpty(self):
        """Return whether this list is empty."""
        return True

class Cons(LispList):
    def __init__(self, hd, tl):
        """Initialize this Cons with head hd and tail tl."""
        self.car = hd
        self.cdr = tl
    def __eq__(self, lst):
        """Return True just when self is structurally equivalent to lst."""
        return isinstance(lst, Cons) and lst.first() == self.first() \
            and lst.tail() == self.tail()
    def __repr__(self):
        """Return a string representing this list."""
        return "Cons(" + repr(self.first()) + ", " + repr(self.tail()) + ")"
    def elements_str(self):
        """Return a string of the elements of self, separated by commas."""
        if self.tail().isEmpty():
            return str(self.first())
        else:
            return str(self.first()) + ", " + self.tail().elements_str()
    def __str__(self):
        """Return a string showing the elements of self."""
        return "[" + self.elements_str() + "]"
    def isEmpty(self):
        """Return whether this list is empty."""
        return False
    def first(self):
        """Return the first element of this list."""
        return self.car
    def tail(self):
        """Return the rest of this list."""
        return self.cdr
```

Figure 1: Code for `LispList`, which is used in some problems.

1. (30 points) [Programming] Define a Python function, `subst(new, old, lst)`, of

   type: (**int**, **int**, LispList(**int**)) -> LispList(**int**)

   that takes an int, `new`, an int `old`, and a LispList of ints, `lst`, and returns a new LispList of ints that is just like `lst` except that each element that was equal to `old` is replaced by `new`.

   The argument `lst` must not be modified at all. Your code must *not* convert the argument to a Python sequence.

   Tests for this problem appear in Figure 2.

```
# $Id: test_subst.py,v 1.1 2019/02/24 22:41:44 leavens Exp $
from LispList import *
from subst import *
def test_subst():
    """Testing for subst."""
    assert subst(9, 1, Nil()) == Nil()
    lst3223 = Cons(3, Cons(2, Cons(2, Cons(3, Nil()))))
    assert subst(7, 3, lst3223) == Cons(7, Cons(2, Cons(2, Cons(7, Nil()))))
    # subst does not change the argument list
    assert lst3223 == Cons(3, Cons(2, Cons(2, Cons(3, Nil()))))
    lst102030 = Cons(10, Cons(20, Cons(30, Nil())))
    assert subst(50, 20, lst102030) == Cons(10, Cons(50, Cons(30, Nil())))
    assert subst(50, 20, lst102030.tail()) == Cons(50, Cons(30, Nil()))
    assert subst(99, 50, lst102030) == lst102030
    # subst does not change the argument list
    assert lst102030 == Cons(10, Cons(20, Cons(30, Nil())))
    lst1To10 = Cons(1, Cons(2, Cons(3, Cons(4, Cons(5, Cons(6, Cons(7, \
                                Cons(8, Cons(9, Cons(10, Nil()))))))))))
    assert subst(-2, 5, lst1To10) == Cons(1, Cons(2, Cons(3, Cons(4, \
               Cons(-2, Cons(6, Cons(7, Cons(8, Cons(9, Cons(10, Nil()))))))))))
```

Figure 2: Tests for `subst`.

---

2. (35 points) [Programming] Define a Python function, `sumReciprocals(lst)` of

   type: `(LispList(number)) -> number`

   that when given a LispList of numbers, `lst`, returns the sum of the reciprocals of each non-zero number in `lst`. Tests for this problem appear in Figure 3.

   Your solution must *not* convert the argument list into a Python sequence.

```
# $Id: test_sumReciprocals.py,v 1.1 2019/02/24 22:41:44 leavens Exp $
from LispList import *
from math import isclose
from sumReciprocals import *
def test_sumReciprocals():
    """Testing for sumReciprocals."""
    assert isclose(sumReciprocals(Nil()), 0)
    assert isclose(sumReciprocals(Cons(1, Cons(0, Cons(-1, Nil())))), \
                   1/1 + 1/-1)
    assert isclose(sumReciprocals(Cons(4, Cons(1, Nil()))), 1/4 + 1/1)
    lst03223 = Cons(0, Cons(3, Cons(2, Cons(2, Cons(3, Nil())))))
    assert isclose(sumReciprocals(lst03223), 1/3 + 1/2 + 1/2 + 1/3)
    # sumReciprocals does not change the argument list
    assert lst03223 == Cons(0, Cons(3, Cons(2, Cons(2, Cons(3, Nil())))))
    assert isclose(sumReciprocals(Cons(20, Cons(10, Cons(20, Cons(30, Nil()))))), \
                   1/20 + 1/10 + 1/20 + 1/30)
```

Figure 3: Tests for `sumReciprocals`.

---

3. (35 points) [Programming] Define a function, `whenBigger(goal, lst)`, of

   type: `(number, list(Number)) -> int`

   that takes a positive number, `goal`, and a (Python) list of numbers, `lst`, and returns the smallest (zero-based) index into `lst` such that the sum of the elements with indexes from 0 to `i`, inclusive, is greater than or equal to `goal`. Your code can assume that the sum of all the numbers in `lst` is at least `goal`.

   Examples appear in Figure 4.

```
# $Id: test_whenBigger.py,v 1.1 2019/02/25 00:28:47 leavens Exp leavens $
from whenBigger import *
def test_whenBigger():
    """Testing for whenBigger"""
    assert whenBigger(1, [1]) == 0
    assert whenBigger(1, [1,1,1,55,99,200]) == 0
    assert whenBigger(17.0, [9.01,3.0,5.01,100.1]) == 2
    assert whenBigger(12.0, [9.01,3.0,5.01,100.1]) == 1
    assert whenBigger(60, list(range(100))) == 11
    assert whenBigger(86, list(range(1000))) == 13
    halves = [1.0,0.5,0.25,0.125,0.0625,0.03125,0.015625,0.0078125,0.00390625]
    assert whenBigger(1.9, halves) == 4
    assert whenBigger(1.99, halves) == 7
    assert whenBigger(2, [-3,-2,-1,0,1,2,3,4,5]) == 7
```

Figure 4: Tests for `whenBigger`.
