CIS640 End term Exam

05/11/2017 @ 2PM

Maximum points: 60 + 9

Duration: 75 minutes

Instructions:

A. Please write your name, KSU ID, and the total number of pages in your solution on top right hand corner of the first page of your solution.
B. Number each page used in your solution.
C. Make sure you sign the roster both before starting the exam and after completing the exam.
D. The exam is closed resource. So, access to any sort of resources (including mobile phones/devices) is not permitted during the exam. Except pen, pencil, and eraser, all resources should be placed under the table.
E. Use of mobile phones during the exam is not allowed. So, please turn them off.
F. Please return the exam with your solution.

Questions:

1. What is the workflow (process) of test-driven development? **(6 points)**
    1. Write a new test (from specs).
    2. Write/Modify code necessary to pass the new test.
    3. Run all tests. If any test fails, goto to step 2.
    4. Refactor the code.
    5. Run tests. If any test fails, goto to step 4.
    6. Repeat steps 1-5 until the tests capture the specifications (desired behaviors) of the UUT
2. Given the following definitions of different kinds of quadrilaterals, define the equivalence classes of these quadrilaterals in terms of testable constraints involving (positive length) sides a, b, c, and d and interior angles i, j, k, and l (both in clockwise order — a, i, b, j, c, k, d, l). **(18 points)**
    A. Parallelogram is a quadrilateral with two pairs of parallel sides.
    B. Rectangle is a quadrilateral with four right angles.
    C. Rhombus is a quadrilateral with four equal length sides.
    D. Square is a rectangle with four equal length sides.
    E. Trapezoid is a quadrilateral with at least one pair of parallel sides.
    F. Quadrilateral is a polygon with four sides (and corners). [None of the above]
    - square: i == k && i == 90 && a == b [3 points]
    - rectangle: i == k && i == 90 && a != b [3 points]
        - includes square [-1 point]
    - rhombus: i == k && i != 90 && a == b [3 points]
        - includes square [-1 point]
    - parallelogram: i == k && i != 90 && a != b [3 points]
        - includes rhombus [-1 point]
        - includes rectangle [-1 point]
    - trapezoid: i != k && (i+j == 180 || j+k == 180 || k+l == 180 || l+i == 180) [3 points]
        - includes parallelogram [-1.5 point]
    - quadrilateral: !(i+j == 180 || j+k == 180 || k+l == 180 || l+i == 180) [3 points]

3. *make_power_set(s)* is a Python function that accepts a list *s* of values and returns power set of the unique values in *s* as a set of *frozensets* (unmodifiable set type in Python). It raises *ValueError* exception if *s* is not a list. The function assumes the values in *s* are immutable.
   A. Identify the properties to test the function. **(5 points)**
      1. Function should Raise ValueError for non-list input. [1 point]
      2. Function should return an value of set type [1 point] in which each element is of frozenset type. [1 point]
      3. Function should return a set of size 2^n where n is the number of unique elements in s. [1 point]
      4. All elements of the returned subsets should be elements of s. [1 point]
      5. All subsets of s should be returned. (Not required)
      6. A set containing empty set should be returned if s is empty. (Not required)
      7. All unique elements of s should be present in exactly 2^{n-1} subsets. (Not required)
   B. Write property-based test suite in Python to test the function. **(11 points)**
   - Test suite cannot use builtin/library functions that generate power sets.
   - A power set of set T is the set of all possible subsets of T (including the empty set and T). For a set with n elements, its power set contains 2^n elements.
   - Test suite does not calculate power sets in any form. **(4 points)**

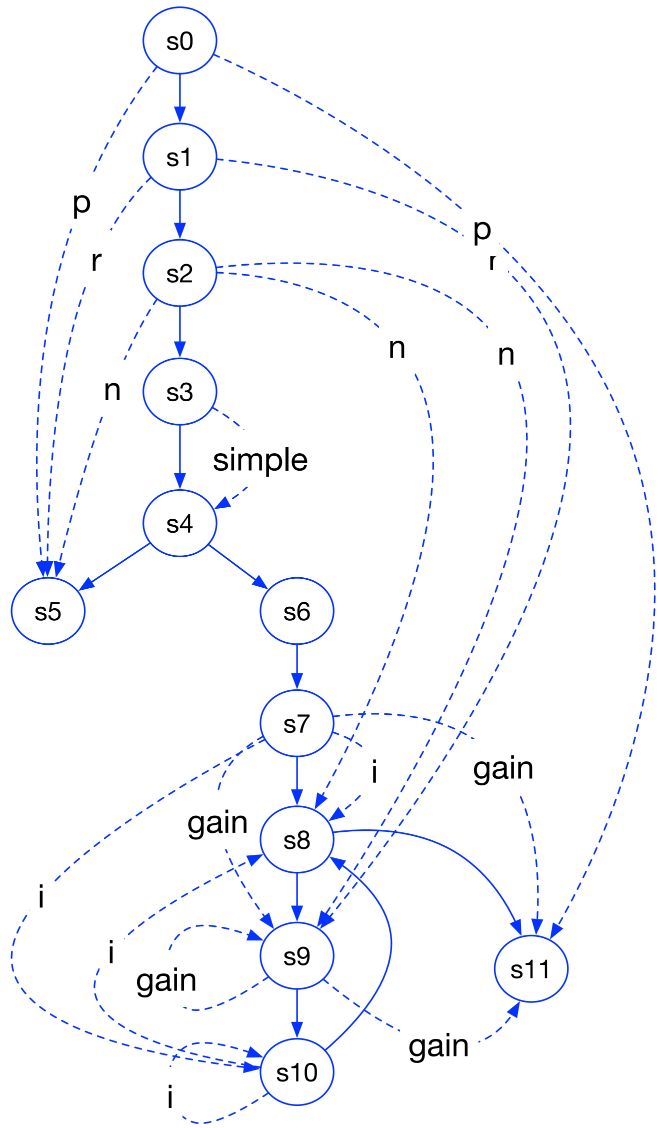
```
# 2pts
@given(st.one_of(st.text(), st.integers(), st.floats(),
st.booleans(), st.none()))
def test_invalid_input_type(values):
    with pytest.raises(ValueError):
        make_power_set(values)

# 3pts
@given(st.lists(st.integers(), max_size=15))
def test_valid_return_type(values):
    power_sets = make_power_set(values)
    assert isinstance(power_sets, set)
    assert all(isinstance(s, frozenset) for s in power_sets)

# 3pts
@given(st.lists(st.integers(), max_size=15))
def test_all_subsets_contain_only_given_values(values):
    power_sets = make_power_set(values)
    for s in power_sets:
       assert s.issubset(values)

# 3pts
@given(st.lists(st.integers(), max_size=15))
def test_powerset_size(values):
    power_sets = make_power_set(values)
    assert len(power_sets) == 2**n
```

4. Construct the CFG along with data flow (def-use) edges of the following Python program. Label def-use edges with corresponding variable. **(14 points)**

```
s0: p = input()
s1: r = input()
s2: n = input()
```

```
 s3: simple = input()
 s4: if simple:
 s5:   return p * r * n
 s6: else:
 s7:   gain, i = 0, 0
 s8:   while i < n:
 s9:     gain *= 1 + r / n
s10:     i += 1
s11:   return p * gain
```

[4 points for CFG,
10 points for DF edges]



5. Identify test inputs for the program in Q4 to achieve 100% coverage of all paths from source to sinks such that every pair of covered paths is mutually distinct — path p1 is mutually distinct from path p2 if set of edges in p1 is not identical to set of edges in p2 — and the set of covered paths is maximal — any path that is mutually distinct from a covered path is also covered. With each test input, list the covered path as a sequence of labels of the statements that form the path. **(6 points)**
   - (p=100, r=0.6, n=1, simple=False): s0, s1, s2, s3, s4, s5
   - (p=100, r=0.6, n=0, simple=True): s0, s1, s2, s3, s4, s6, s7, s8, s11
   - (p=100, r=0.6, n=1, simple=True): s0, s1, s2, s3, s4, s6, s7, s8, s9, s10, s8, s11
6. *is_jolly_jumper(nums)* is a Python function that accepts a list of integers *nums* and returns *True* if *nums* is a jolly jumper; *False*, otherwise. A sequence of n > 0 integers is a jolly jumper if the absolute values of the differences between successive elements take on all possible values 1 through n–1. For example, [1, 4, 2, 3] is a jolly jumper. By definition, any sequence of a single integer is a jolly jumper. The function assumes *nums* is a non-empty list of integers. Write property-based test suite in Python to test the function. **(5 points)**

```
# 1 point
@given(st.lists(st.integers(), min_size=1, max_size=1))
def test_one_element_sequence_is_a_jolly_jumper(values):
    assert is_jolly_jumper(values)

# 2 points
@given(st.lists(st.integers(), min_size=2, max_size=15))
@example([101, 104, 102, 103])
def
test_not_jolly_jumper_if_max_difference_is_not_equal_to_len_of_va
lues(values):
    if (max(values) - min(values) != len(values) - 1):
        assert not is_jolly_jumper(values)

# 5 points
@given(st.lists(st.integers(), min_size=2, max_size=15))
def test_jolly_jumper(values):
    tmp1 = sorted([abs(values[i-1] - values[i]) for i in range(1,
len(values))])
    assert (tmp1 == range(1, len(values))) ==
is_jolly_jumper(values)
```