CS 61A Structure and Interpretation of Computer Programs
Summer 2017

Quiz 5

INSTRUCTIONS

- You have 10 minutes to complete this quiz.
- The exam is closed book, closed notes, closed computer, closed calculator.
- The final score for this quiz will be assigned based on **effort** rather than correctness.
- Mark your answers **on the exam itself**. We will *not* grade answers written on scratch paper.
- For multiple choice questions,
  - ☐ means mark **all options** that apply
  - ◯ means mark a **single choice**

Last name	
First name	
Student ID number	
CalCentral email (`_@berkeley.edu`)	
Teaching Assistant	◯ Alex Stennet ◯ Kelly Chen ◯ Angela Kwon ◯ Michael Gibbes ◯ Ashley Chien ◯ Michelle Hwang ◯ Joyce Luong ◯ Mitas Ray ◯ Karthik Bharathala ◯ Rocky Duan ◯ Kavi Gupta ◯ Samantha Wong
Name of the person to your left	
Name of the person to your right	
All the work on this exam is my own. **(please sign)**	

1. (5 points) Return of the Jedi

Let's implement a data abstraction for basketball players. Our constructor takes in a name, a position (1, 2, 3, 4, or 5), and, optionally, a backup position. Our selectors retrieve information about a player.

```
def player(name, position, backup=None):  # if no backup position, default to None
    return {'name': name, 'position': position, 'backup': backup}

def name(player):
    return player['name']
def position(player):
    return player['position']
def backup(player):
    return player['backup']
```

```
def insert(lst, elem):
    """Add elem to lst if elem is not already contained in lst.
    >>> insert(insert([1, 2, 3], 5), 2)
    [1, 2, 3, 5]
    """
    return lst if elem in lst else lst + [elem]
```

When we make a basketball team, we want to make sure that there is at least one player for each position. So we define a function `check_team` that takes in a non-empty list of players. `check_team` returns `True` if there is at least one player per position, and `False` otherwise.

The following implementation works, but it breaks abstraction barriers! Fill in the square to the left of each line that breaks an abstraction barrier. Then, cross out each violation and, above the original expression, write some replacement code that has no violations and maintains correctness.

```
def check_team(players):
    """Make sure there is at least one player per position.

    >>> check_team([player('Steph', 1),   player('KD', 3, 4),   player('Klay', 2),
    ...             player('Iggy', 4, 3), player('Dray', 4, 5)])
    True
    >>> check_team([player('LeBron', 3, 4), player('Kyrie', 1), player('Love', 4, 5)]
    False
    """
    def checker(players, covered):

☐       if len(covered) == 5:

☐           return True

☐       elif len(players) == 0:

☐           return False

☐       p = players[0]

☐       in_main_role = checker(players[1:], insert(covered, p['position']))

☐       if p['backup'] != None:

☐           in_backup_role = checker(players[1:], insert(covered, p['backup']))

☐           return in_main_role or in_backup_role

☐       return in_main_role

☐   return checker(players, [])
```