# Homework 4: Scope and Formal Semantics

Due October 4th at 10pm

There are two steps to submitting this assignment:

(1) Place your write-up in a Google Drive folder that is shared with me.

(2) Fill out the Homework 4 Submission Form.

## Part 1: Racket lists (35 points)

In Lecture 3, we learned that lists are defined recursively in Racket: they are either null, or a pair whose second item is itself a list. The empty list (null) is a value: it cannot be evaluated any further.

### 1. lists as values (10 points)

Does Racket allow lists to contain expressions that are not values?

Provide a table of observations that you used to reach your conclusion.

### 2. appending (20 points)

Python has two methods that are often interchangeably called append. One is +, which is an overloaded concatenation operator. The second is a list method that takes one argument and inserts it at the tail of the list.

Racket also has two append-like operators. The first, **append**, is used to combine lists. It takes any number of arguments. The second, **cons**, takes two arguments and produces a pair.

(Since our subset of Racket does not have mutation, neither can modify an existing list.)

These Racket operators have one funny property: they do not always return lists. If their last argument is not a list, they return an *improper list*, indicated with a period between the head and tail of the pair:

```
(cons 5 null)
> '(5)

(cons 5 (cons 6 null))
> '(5 6)

(cons 5 6)
> '(5 . 6)
```

```
(append null null)
> '()

(append '(5) null))
> '(5)

(append '(5) 6)
> '(5 . 6)
```

**Pick one of these operators and write a big step semantics for it.** Make sure to test your semantics so that it accurately reflects the operator's behavior.

### 3. append and improper lists (5 points)

Consider a version of append called **proper-append** that is guaranteed to return a proper list (i.e., throws an error if its arguments are not proper lists). Think through the recursion involved in terms of efficiency. Do you think **proper-append** can be as efficient as Racket's **append**? Why or why not?

## Part 2: Scope fieldwork (65 points)

In this set of questions, you'll explore the scope of a familiar programming language (Python) using the observation-based approach we took in class.

For each question, please record a table of observations that you have made. **Please make sure your observations are reproducible.** Because Python allows mutation, you may need to include more details about the program context in your observation table.

### 4. variable declarations and updates (20 points)

In Python, the variable definition operator = is **overloaded**: it can be used both to define a variable and to update its value.

Explore the scope of the Python variable definition operator. You should explore its behavior in at least the following contexts: (1) the global program context, (2) within a function, and (3) within nested functions. You should also explore how multiple uses of the operator interact.

A. Construct an observation table like we did in class.

B. State a hypothesis about the scope of the variable definition operator. You need not formalize it in big step semantics, but try to make it as precise as possible using the terminology we have learned so far.

### 6. global (20 points)

Python's **global** declaration modifies the scope of variables within its scope.

```
def foo():
    global x
    x = "outside"
    return()
foo()
x
```

What does **global** do?

A. Construct an observation table like we did in class. You should try to discover (1) what **global** does to variables within its scope, (2) what **global**'s scope is, and (3) how **global** interacts with mutation.

B. State a hypothesis about the semantics of the **global** declaration. You need not formalize it in big step semantics, but try to make it as precise as possible using the terminology we have learned so far.

## 7. nonlocal (20 points)

Python 3.0 added a second kind of scope declaration: **nonlocal**. An example use is shown below:

```
def foo():
    x = "outside"
    def goo():
        nonlocal x
        x = "inside"
        return x
    return (goo(),x)

foo()
```

What does **nonlocal** do?

A. Construct an observation table like we did in class. You should try to discover (1) what **nonlocal** does to variables within its scope, (2) what **nonlocal**'s scope is, and (3) how **nonlocal** interacts with mutation.

B. State a hypothesis about the semantics of the **nonlocal** declaration. You need not formalize it in big step semantics, but try to make it as precise as possible using the terminology we have learned so far.

## 8. locals (5 points)

Python also provides an operator called **locals**, which returns a dictionary of the variable bindings in the current scope.

```
def foo(x):
    y = "cat"
    return locals()
foo(7)                              # produces: {'x': 7, 'y': 'cat'}
```

Can this be modeled using the substitution model of variable binding? Why or why not?

# Extra Credit

## 1. what's the hurry?

In our big step semantics for function application, we stipulated that both expressions had to first be evaluated to values. For $e_1$, this was important, because we needed to make sure that it was, in

fact, a function. But is it also important to evaluate $e_2$ to value? Explain why you think this is or isn't necessary, giving example derivations to support your argument.

## 2. steamroll

Write a version of flatten that takes a nested, potentially improper list and returns a list with all items within.

## 3. wacky revisited

In Homework 2, we asked you to consider the wacky language shown in the code snippet below.

```
def foo():
    x = 5
    return
foo()
print(x)                # prints 5
```

Can you derive this behavior with any of the Python elements that you explored in Section 2? You may place this code snippet inside a larger program, and add uses of **nonlocal**, **locals**, and **global** within it as needed.