Lab 10 Solutions **lab10.zip (lab10.zip)**

Solution Files

Topics

Consult this section if you need a refresher on the material for this lab. It's okay to skip directly to the questions and refer back here should you get stuck.

Scheme

Scheme is a famous functional programming language from the 1970s. It is a dialect of Lisp (which stands for LISt Processing). The first observation most people make is the unique syntax, which uses a prefix notation and (often many) nested parentheses (see http://xkcd.com/297/ (http://xkcd.com/297/)). Scheme features first-class functions and optimized tail-recursion, which were relatively new features at the time.

> Our course uses a custom version of Scheme (which you will build for Project 4) included in the starter ZIP archive. To start the interpreter, type `python3 scheme`. To run a Scheme program interactively, type `python3 scheme -i <file.scm>`. To exit the Scheme interpreter, type `(exit)`. You may find it useful to try code.cs61a.org/scheme (https://code.cs61a.org/scheme) when working through problems, as it can draw environment and box-and-pointer diagrams and it lets you walk your code step-by-step (similar to Python Tutor). Don't forget to submit your code through Ok though!

Scheme Editor

As you're writing your code, you can debug using the Scheme Editor. In your `scheme` folder you will find a new editor. To run this editor, run `python3 editor`. This should pop up a window in your browser; if it does not, please navigate to localhost:31415 (localhost:31415)

and you should see it.

Make sure to run `python3 ok` in a separate tab or window so that the editor keeps running.

If you find that your code works in the online editor but not in your own interpreter, it's possible you have a bug in code from an earlier part that you'll have to track down. Every once in a while there's a bug that our tests don't catch, and if you find one you should let us know!

Expressions

Control Structures

Lists

Defining Names

Lambda Functions

Required Questions

What Would Scheme Display?

Q1: Combinations

Let's familiarize ourselves with some built-in Scheme procedures and special forms!

> Use Ok to unlock the following "What would Scheme print?" questions:
>
> ```
> python3 ok -q combinations -u
> ```

```
scm> (- 10 4)

scm> (* 7 6)

scm> (+ 1 2 3 4)

scm> (/ 8 2 2)

scm> (quotient 29 5)

scm> (modulo 29 5)
```

```
scm> (= 1 3)                    ; Scheme uses '=' instead of '==' for comparison

scm> (< 1 3)

scm> (or 1 #t)                  ; or special form short circuits

scm> (and #t #f (/ 1 0))

scm> (not #t)
```

```
scm> (define x 3)

scm> x

scm> (define y (+ x 4))

scm> y

scm> (define x (lambda (y) (* y 2)))

scm> (x y)
```

```
scm> (if (not (print 1)) (print 2) (print 3))

scm> (* (if (> 3 2) 1 2) (+ 4 5))

scm> (define foo (lambda (x y z) (if x y z)))

scm> (foo 1 2 (print 'hi))

scm> ((lambda (a) (print 'a)) 100)
```

Coding Questions

Q2: Over or Under

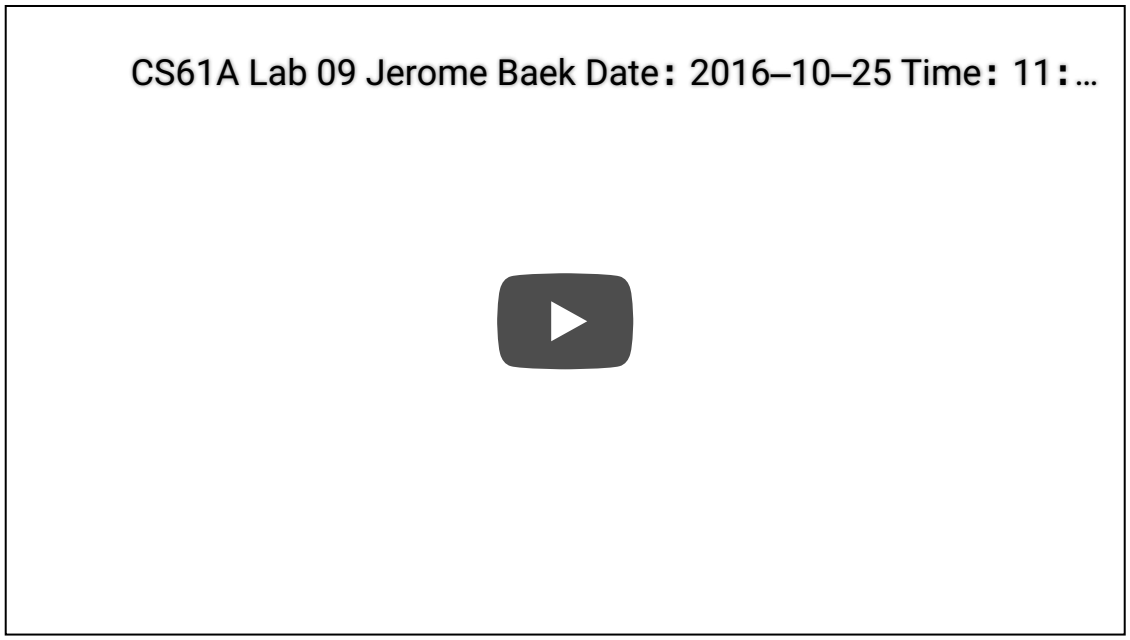
Define a procedure `over-or-under` which takes in a number `num1` and a number `num2` and returns the following:

- -1 if `num1` is less than `num2`
- 0 if `num1` is equal to `num2`
- 1 if `num1` is greater than `num2`

> Challenge: Implement this in 2 different ways using `if` and `cond`!

```
(define (over-or-under num1 num2)
  (cond
    ((< num1 num2) -1)
    ((= num1 num2) 0)
    (else 1))
)
```

Video walkthrough:



YouTube link (https://youtu.be/UJ37SCaM3cQ?t=35m46s)

Use Ok to test your code:

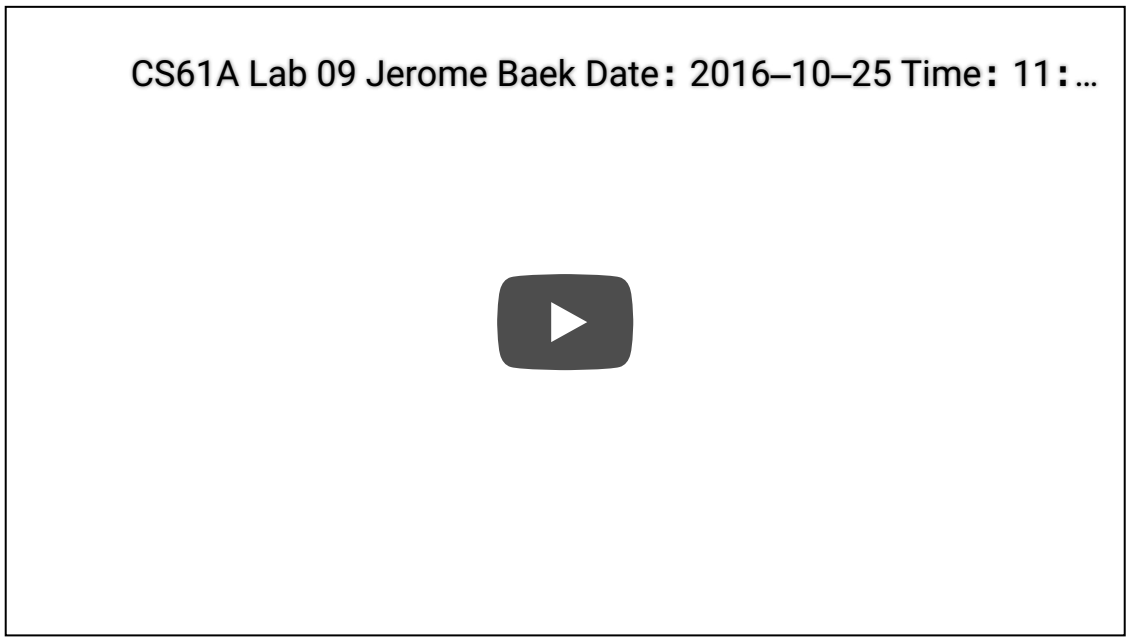
```
python3 ok -q over_or_under
```

Q3: Make Adder

Write the procedure `make-adder` which takes in an initial number, `num`, and then returns a procedure. This returned procedure takes in a number `inc` and returns the result of `num + inc`.

> *Hint*: To return a procedure, you can either return a `lambda` expression or `define` another nested procedure. Remember that Scheme will automatically return the last clause in your procedure.
>
> You can find documentation on the syntax of `lambda` expressions in the 61A scheme specification! (https://cs61a.org/articles/scheme-spec/#lambda)

```
(define (make-adder num)
  (lambda (inc) (+ inc num))
)
```



YouTube link (https://youtu.be/UJ37SCaM3cQ?t=47m4s)

Use Ok to test your code:

```
python3 ok -q make_adder
```

Q4: Compose

Write the procedure `composed`, which takes in procedures `f` and `g` and outputs a new procedure. This new procedure takes in a number `x` and outputs the result of calling `f` on `g` of `x`.

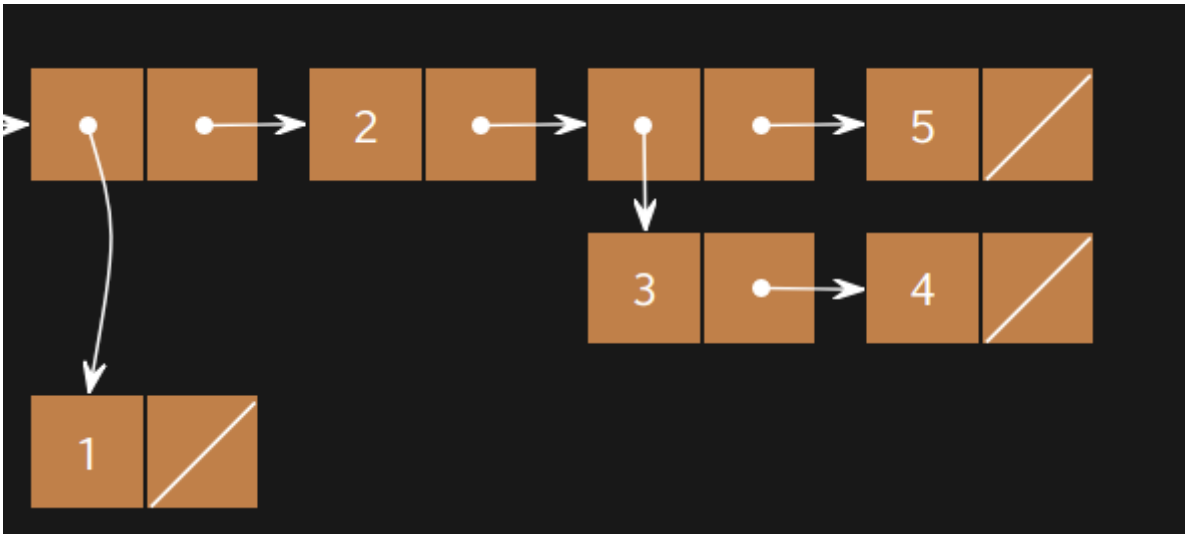
```
(define (composed f g)
  (lambda (x) (f (g x)))
)
```

Use Ok to test your code:

```
python3 ok -q composed
```

Q5: Make a List

In this problem you will create the list with the following box-and-pointer diagram:



> Challenge: try to create this list in multiple ways, and using multiple list constructors!

```
(define lst
  (cons (cons 1 nil)
        (cons 2
              (cons (cons 3 (cons 4 nil))
                    (cons 5 nil))))
)
```

Use Ok to unlock and test your code:

```
python3 ok -q make_structure -u
python3 ok -q make_structure
```

Submit

Make sure to submit this assignment by running:

```
python3 ok --submit
```

Optional Questions

Q6: Remove

Implement a procedure `remove` that takes in a list and returns a new list with *all* instances of `item` removed from `lst`. You may assume the list will only consist of numbers and will not have nested lists.

> *Hint*: You might find the built-in `filter` procedure useful (though it is definitely possible to complete this question without it).
>
> You can find information about how to use `filter` in the 61A Scheme builtin specification (https://cs61a.org/articles/scheme-builtins/#pair-and-list-manipulation)!

```
(define (remove item lst)
  (cond ((null? lst) '())
        ((equal? item (car lst)) (remove item (cdr lst)))
        (else (cons (car lst) (remove item (cdr lst))))))
)

(define (remove item lst)
  (filter (lambda (x) (not (= x item))) lst))
```

Use Ok to unlock and test your code:

```
python3 ok -q remove -u
python3 ok -q remove
```