Lab 9 Solutions **lab09.zip (lab09.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 scheme.cs61a.org (https://scheme.cs61a.org) 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!

Expressions

Control Structures

Lists

Defining procedures

Lambdas

Required Questions

What Would Scheme Display?

Q1: WWSD: Lists

Use Ok to test your knowledge with the following "What Would Scheme Display?" questions:

```
python3 ok -q wwsd_lists -u
```

```
scm> (cons 1 (cons 2 nil))
------

scm> (car (cons 1 (cons 2 nil)))
------

scm> (cdr (cons 1 (cons 2 nil)))
------

scm> (list 1 2 3)
------

scm> '(1 2 3)
------

scm> (cons 1 '(list 2 3))  ; Recall quoting
------
```

```
scm> '(cons 4 (cons (cons 6 8) ()))
(cons 4 (cons (cons 6 8) ()))
```

Coding Questions

Q2: Over or Under

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

- -1 if `a` is less than `b`
- 0 if `a` is equal to `b`
- 1 if `a` is greater than `b`

```
(define (over-or-under a b)
  (cond
    ((< a b) -1)
    ((= a b) 0)
    (else 1))
)

;;; Tests
(over-or-under 1 2)
; expect -1
(over-or-under 2 1)
; expect 1
(over-or-under 1 1)
; expect 0
```

Video walkthrough: https://youtu.be/UJ37SCaM3cQ?t=35m46s (https://youtu.be/UJ37SCaM3cQ?t=35m46s)

Use Ok to unlock and test your code:

```
python3 ok -q over_or_under -u
python3 ok -q over_or_under
```

Q3: Filter Lst

Write a procedure `filter-lst`, which takes a predicate `fn` and a list `lst`, and returns a new list containing only elements of the list that satisfy the predicate. The output should contain the elements in the same order that they appeared in the original list.

```
(define (filter-lst fn lst)
  (cond ((null? lst) '())
        ((fn (car lst)) (cons (car lst) (filter-lst fn (cdr lst))))
        (else (filter-lst fn (cdr lst))))
)

;;; Tests
(define (even? x)
  (= (modulo x 2) 0))
(filter-lst even? '(0 1 1 2 3 5 8))
; expect (0 2 8)
```

Video walkthrough: https://youtu.be/UJ37SCaM3cQ?t=39m39s (https://youtu.be/UJ37SCaM3cQ?t=39m39s)

Use Ok to unlock and test your code:

```
python3 ok -q filter_lst -u
python3 ok -q filter_lst
```

Q4: Make Adder

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

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.

```
(define (make-adder n)
  (lambda (x) (+ x n))
)

;;; Tests
(define adder (make-adder 5))
(adder 8)
; expect 13
```

Video walkthrough: https://youtu.be/UJ37SCaM3cQ?t=47m4s (https://youtu.be/UJ37SCaM3cQ?t=47m4s)
Use Ok to unlock and test your code:

```
python3 ok -q make_adder -u
python3 ok -q make_adder
```

Submit

Make sure to submit this assignment by running:

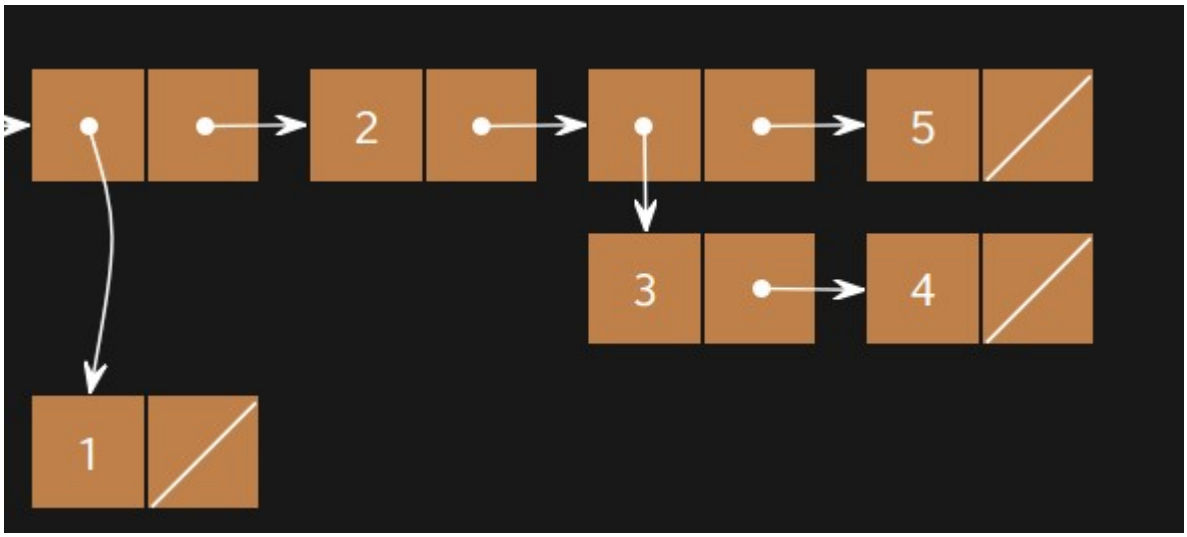
```
python3 ok --submit
```

Optional Questions

> The following questions are for **extra practice**. However, they are in scope and we recommend you do them if you have time.

Q5: Make a List

Create the list with the following box-and-pointer diagram:




```
(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
```

Q6: 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 unlock and test your code:

```
python3 ok -q composed -u
python3 ok -q composed
```

Q7: 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 `filter-lst` procedure useful.

```
(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))

;;; Tests
(remove 3 nil)
; expect ()
(remove 3 '(1 3 5))
; expect (1 5)
(remove 5 '(5 3 5 5 1 4 5 4))
; expect (3 1 4 4)
```

Use Ok to unlock and test your code:

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

Q8: No Repeats

Implement `no-repeats`, which takes a list of numbers `s` as input and returns a list that has all of the unique elements of `s` in the order that they first appear, but no repeats. For example, `(no-repeats (list 5 4 5 4 2 2))` evaluates to `(5 4 2)`.

Hints: To test if two numbers are equal, use the `=` procedure. To test if two numbers are not equal, use the `not` procedure in combination with `=`. You may find it helpful to use the `filter-lst` procedure.

```
(define (no-repeats s)
  (if (null? s) s
    (cons (car s)
      (no-repeats (filter (lambda (x) (not (= (car s) x))) (cdr s))))))
)
```

Use Ok to unlock and test your code:

```
python3 ok -q no_repeats -u
python3 ok -q no_repeats
```

Q9: Substitute

Write a procedure `substitute` that takes three arguments: a list `s`, an `old` word, and a `new` word. It returns a list with the elements of `s`, but with every occurrence of `old` replaced by `new`, even within sub-lists.

Hint: The built-in `pair?` predicate returns True if its argument is a `cons` pair.

Hint: The `=` operator will only let you compare numbers, but using `equal?` or `eq?` will let you compare symbols as well as numbers. For more information, check out the Scheme Built-in Procedure Reference (/articles/scheme-builtins.html#boolean-operations).

Use Ok to unlock and test your code:

```
python3 ok -q substitute -u
python3 ok -q substitute
```

```
(define (substitute s old new)
  (cond ((null? s) nil)
        ((pair? (car s)) (cons (substitute (car s) old new)
                               (substitute (cdr s) old new)))
        ((equal? (car s) old) (cons new
                                    (substitute (cdr s) old new)))
        (else (cons (car s) (substitute (cdr s) old new))))
)
```

Remember that we want to use `equal?` to compare symbols since `=` will only work for numbers!

If the input list is empty, there's nothing to substitute. That's a pretty straightforward base case. Otherwise, our list has at least one item.

We can break the rest of this problem into roughly two big cases:

Nested list here

This means that `(car s)` is a pair. In this case, we have to dig deeper and recurse on `(car s)` since there could be symbols to replace in there. We must recurse on `(cdr s)` as well to handle the rest of the list.

No nested list here

This is the case if `(car s)` is *not* a pair. Of course, there could still be nesting later

on in the list, but we'll rely on our recursive call to handle that.

If `(car s)` matches `old`, we'll make sure to use `new` in its place. Otherwise, we'll use `(car s)` without replacing it. In both cases, we must recurse on the rest of the list.

Video walkthrough: https://youtu.be/LAr-_twxXao (https://youtu.be/LAr-_twxXao)

Q10: Sub All

Write `sub-all`, which takes a list `s`, a list of `old` words, and a list of `new` words; the last two lists must be the same length. It returns a list with the elements of `s`, but with each word that occurs in the second argument replaced by the corresponding word of the third argument. You may use `substitute` in your solution. Assume that `olds` and `news` have no elements in common.

```
(define (sub-all s olds news)
  (if (null? olds)
    s
    (sub-all (substitute s (car olds) (car news))
             (cdr olds)
             (cdr news)))
)
```

Solving `sub-all` means just repeatedly doing the substitute operation we wrote earlier. If this were Python, we might iterate over the list of olds and news, calling `substitute` and saving the result for the next call. For example, it might look something like this:

```
def sub_all(s, olds, news):
    for o, n in zip(olds, news):
        s = substitute(s, o, n)
    return s
```

If that approach makes sense, then the only tricky part now is to translate that logic into Scheme. Since we don't have access to iteration, we have to reflect our progress in our recursive call to `sub-all`. This means shortening the list of `olds` and `news`, and most importantly, feeding the result of `substitute` into the next call.

Therefore, the base case is if we have nothing we want to replace in `olds`. Checking if `news` is empty would also be ok, since `olds` and `news` should be the same length.

What doesn't work, however, is checking if `s` is an empty list. While this won't make your solution wrong on its own, it's an **insufficient** base case. Remember that we're passing in `s` with elements replaced into the recursive call, which means it won't become any shorter.

Video walkthrough: https://youtu.be/avYPLlaBtj0 (https://youtu.be/avYPLlaBtj0)

Use Ok to unlock and test your code:

```
python3 ok -q sub_all -u
python3 ok -q sub_all
```

CS 61A (/)

Weekly Schedule (/weekly.html)

Office Hours (/office-hours.html)

Staff (/staff.html)

Resources (/resources.html)

Studying Guide (/articles/studying.html)

Debugging Guide (/articles/debugging.html)

Composition Guide (/articles/composition.html)

Policies (/articles/about.html)

Assignments (/articles/about.html#assignments)

Exams (/articles/about.html#exams)

Grading (/articles/about.html#grading)