Lab 10: Scheme **lab10.zip (lab10.zip)**

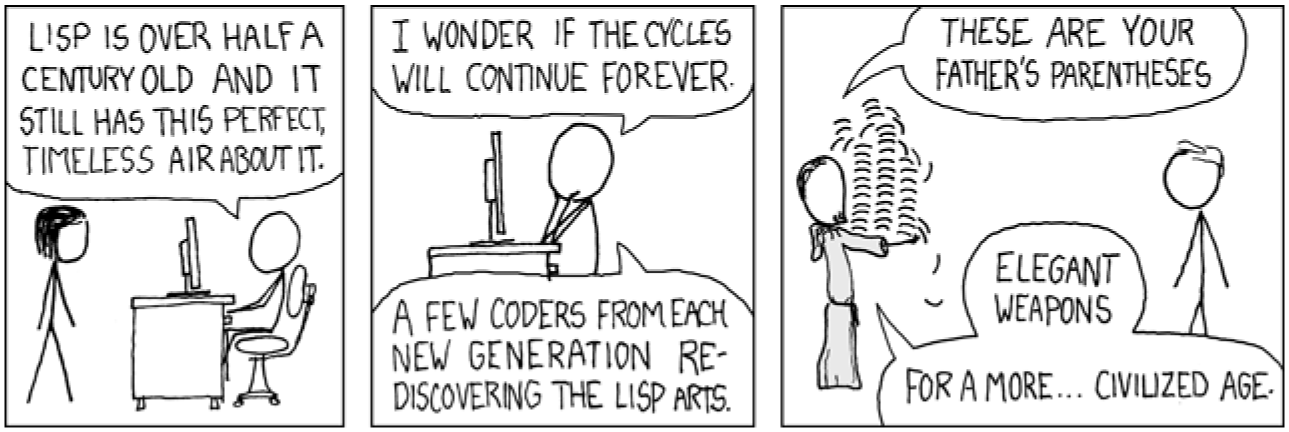
Due by 11:59pm on Wednesday, November 1.

Starter Files

Download lab10.zip (lab10.zip). Inside the archive, you will find starter files for the questions in this lab, along with a copy of the Ok (ok) autograder.

Scheme Introduction

Scheme is a famous functional programming language from the 1970s. It is a dialect of Lisp (which stands for LISt Processing). The syntax of Scheme is very unique: it involves prefix notation and many nested parentheses (see http://xkcd.com/297/ (http://xkcd.com/297/)). Scheme features first-class functions and optimized tail-recursion, which were fairly new when Scheme was introduced.

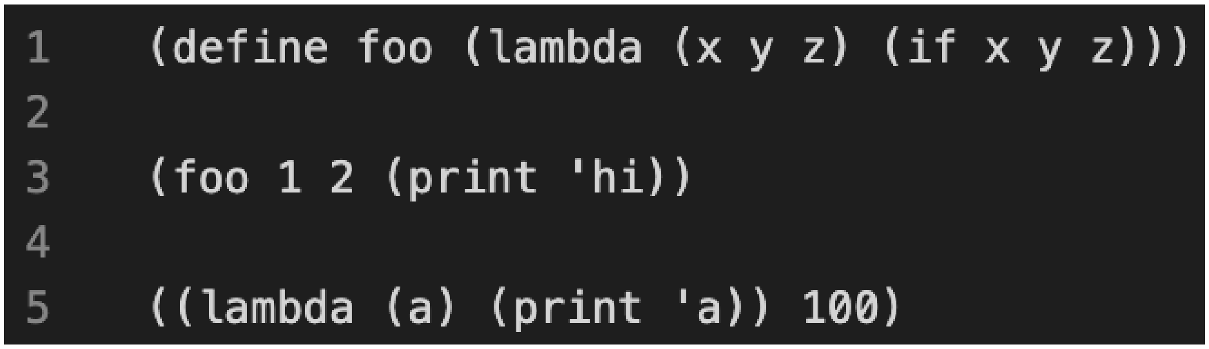

LISP IS OVER HALF A CENTURY OLD AND IT STILL HAS THIS PERFECT, TIMELESS AIR ABOUT IT.
I WONDER IF THE CYCLES WILL CONTINUE FOREVER.
A FEW CODERS FROM EACH NEW GENERATION RE-DISCOVERING THE LISP ARTS.
THESE ARE YOUR FATHER'S PARENTHESES
ELEGANT WEAPONS
FOR A MORE... CIVILIZED AGE.

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)`.

Recommended VS Code Extensions

If you use VS Code as your text editor, we have found these extensions to be quite helpful for Scheme :). You only need to install one because both do syntax coloring!

Before:

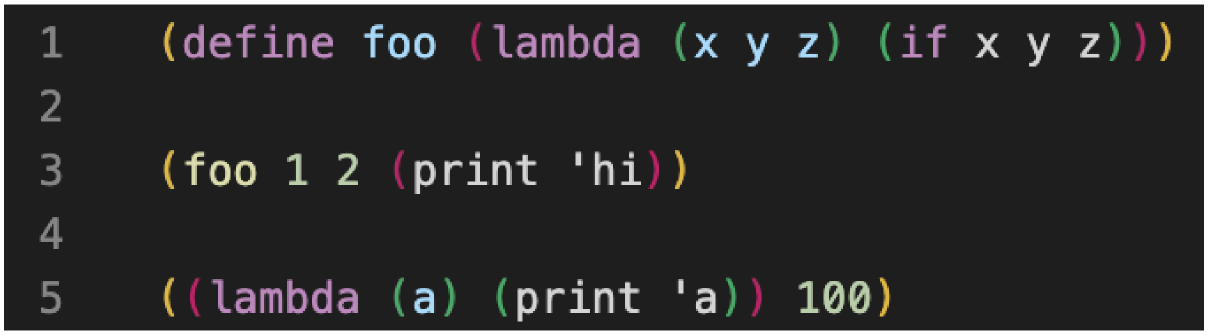


```
(define foo (lambda (x y z) (if x y z)))

(foo 1 2 (print 'hi))

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

After:



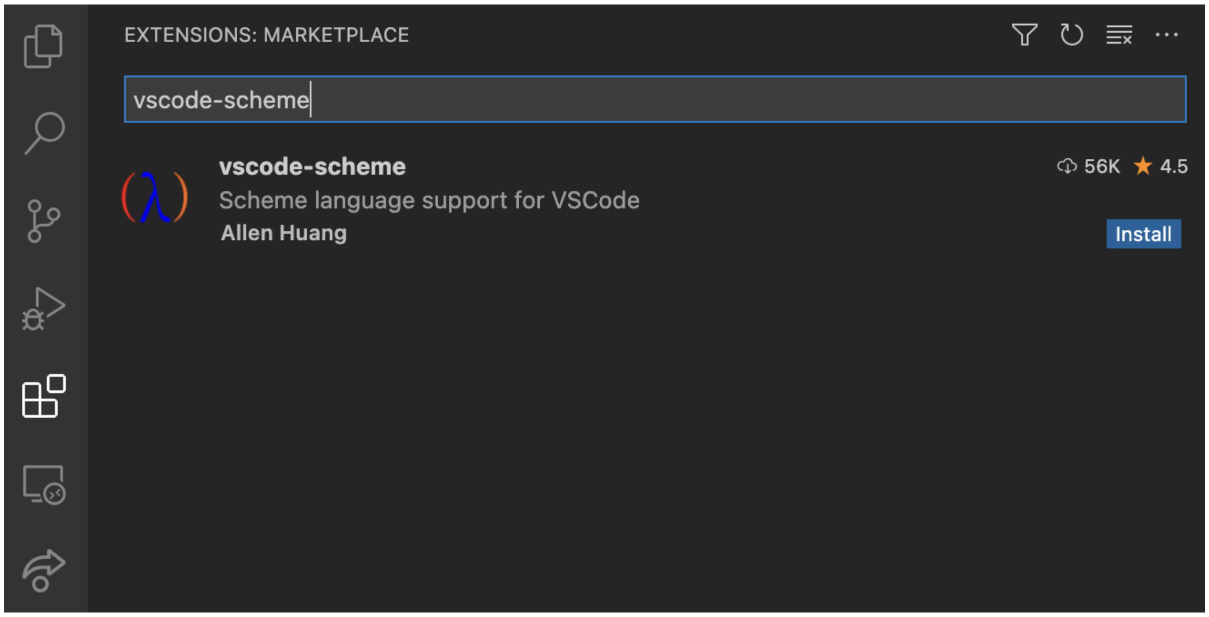
```
(define foo (lambda (x y z) (if x y z)))

(foo 1 2 (print 'hi))

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

Extensions:

vscode-scheme (https://marketplace.visualstudio.com/items?itemName=sjhuangx.vscode-scheme)



Scheme Editor

You can write your code by opening the designated `.scm` file in your text editor.

You can also type directly into the 61A Scheme Editor, which provides testing and debugging tools. To open the 61A Scheme Editor, run `python3 editor` inside the `hw07` folder. This should pop up a window in your browser; if it does not, please navigate to localhost:31415 (localhost:31415) while `python3 editor` is running and you should see it.

If you choose to code in the 61A Scheme Editor, don't forget to save your work before running Ok tests and before closing the editor. Make sure to run `python3 ok` in a separate tab or window so that the editor keeps running. To stop running the editor and return to the command line, type `Ctrl-C`.

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 your 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!

You may find code.cs61a.org/scheme (https://code.cs61a.org/scheme) useful. This Scheme interpreter can draw environment and box-and-pointer diagrams. It also lets you walk through your code step-by-step, like Python Tutor. But don't forget to submit your code to the appropriate Gradescope assignment!

Required Questions

Getting Started Videos

Expressions

Consult the drop-downs below if you need a refresher on Expressions. It's okay to skip directly to the questions and refer back here should you get stuck.

Primitive Expressions

Call Expressions

Special Forms

Consult the drop-down if you need a refresher on Special Forms. It's okay to skip directly to the questions and refer back here should you get stuck.

Special Forms

Q1: WWSD: 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)
```

Scheme Functions

Consult the drop-down if you need a refresher on Scheme Functions. It's okay to skip directly to the questions and refer back here should you get stuck.

Defining Functions

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)
  'YOUR-CODE-HERE
)
```

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)
  'YOUR-CODE-HERE
)
```

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)
  'YOUR-CODE-HERE
)
```

Use Ok to test your code:

```
python3 ok -q composed
```

Q5: Repeat

Write the procedure `repeat`, which takes in a procedure `f` and a number `n`, and outputs a new procedure. This new procedure takes in a number `x` and outputs the result of applying `f` to `x` a total of `n` times. For example:

```
scm> (define (square x) (* x x))
square
scm> ((repeat square 2) 5) ; (square (square 5))
625
scm> ((repeat square 3) 3) ; (square (square (square 3)))
6561
scm> ((repeat square 1) 7) ; (square 7)
49
```

> *Hint:* The `composed` function you wrote in the previous problem might be useful.

```
(define (repeat f n)
  'YOUR-CODE-HERE
)
```

Use Ok to test your code:

```
python3 ok -q repeat
```

Optional Questions

Q6: Greatest Common Divisor

The GCD is the the greatest common divisor of two numbers.

Write the procedure `gcd`, which computes the GCD of numbers `a` and `b`. Recall that Euclid's Algorithm tells us that the GCD of two values is either of the following:

- the smaller value if it evenly divides the larger value, or
- the greatest common divisor of the smaller value and the remainder of the larger value divided by the smaller value

In other words, if `a` is greater than `b` and `a` is not divisible by `b`, then

```
gcd(a, b) = gcd(b, a % b)
```

You may find the provided procedures `min` and `max` helpful. You can also use the built-in `modulo` and `zero?` procedures.

```
scm> (modulo 10 4)
2
scm> (zero? (- 3 3))
#t
scm> (zero? 3)
#f
```

```
(define (max a b) (if (> a b) a b))
(define (min a b) (if (> a b) b a))
(define (gcd a b)
  'YOUR-CODE-HERE
)
```

Use Ok to test your code:

```
python3 ok -q gcd
```