Worksheet 8

Scheme

We are going to be learning a new programming language called **Scheme**

Let's start with the basics

Primitives

Also called atomic primitive expressions, these are usually numbers or Booleans.

Eg.

```
scm> 61
61
scm> 61.1
61.1
scm> #t
#t
```

Tip: #t and #f are the True and False values in Scheme

Procedures

What is called a function in Python, is called a procedure in Scheme.

If we wanted to define a variable, we would do something like this:

```
scm> (define x 5)
x
scm> x
5
```

Here, we assigned the value 5 to the variable named 'x'.

As you may have noticed, the procedure 'define' returns the name of the variable that was defined.

Let's try defining a function.

```
scm> (define (same x) x)
same
scm> (same 2)
2
```

As you see, it is slightly different from just assigning a variable to a value. In the inner parenthesis (same x), 'same' refers to the name of the function, while 'x' refers to the argument name. We can add as many arguments as are needed, just like in Python.

This function just returns whatever is inputted. When we define a function, after defining the name and the argument names, we *immediately* return what the function should return. In this case, we return the value of x.

In Scheme, to call functions, we use parenthesis to define the operator we want to call, and the operands we want to call the function on.

In the example above, we saw 'same' as the operator, and the argument 2 as the operand.

Let's look at some different functions:

```
scm> (+ 3 4)
7
scm> (* (- 4 5) 9)
-9
```

These are the built-in arithmetic functions.

As you can see, we have the operator first, and then the operands following. We evaluate the operator, then each operand before applying the operator on the operands.

Tip:

```
scm> (+ 1 2 5 9)
17
```

Is the same as

```
1 + 2 + 5 + 9
```

In Python

Practice

WWSD (What Would Scheme Display?)

```
scm> (+ 5 (- 2 3))


scm> (define x (- 3 (/ 4 2)))


scm> (* x x x 5 x)


scm> (define (y z a b c) (* (/ (+ (- c b) a) 1) z))


scm> (- 90 100 (+ (y 5 2 1 8) 30))


scm> (+ 3 (define t 5))


scm> (define x (define t 5))


scm> (+ x 5)


scm> x
```

Special Forms

Let's look at a few special cases

If Statements:

In Scheme, we only have the condition, and a TRUE or FALSE case for the condition.

Let's look at some examples:

```
scm> (if #t (* 43 2) (/ 43 4))
86
scm> (if (> 43 44) 99 100)
100
```

Based on the condition's Boolean value, we choose to run a certain procedure.

Boolean Statements

Just like Python, Scheme has the *and/or/not* expressions.

Examples:

```
scm> (or #t (/ 1 0))
#t
scm> (and 1 #f)
#f
scm> (and 1 2)
2
scm> (and 0 #t) ;0 is a truthy value in Scheme
#t
scm> (not (< 4 3))
#t
```

Practice

```
scm> (if (- 4 (+ 1 3)) 99 1)


scm> (if (and #t (or 2 3) (if (not (< 3 2)) (= 5 65) (/ 3 2))) (/ 1 0) 4)
```

Lambda Expressions

We can also create lambda expressions in Scheme. Remember in Python, lambda functions are functions without a name. Similarly, we can construct lambda functions almost exactly the same was as we use define to construction functions in Scheme, but without a name.

```
scm> (lambda (x) (* x 2))
(lambda (x) (* x 2))
```

Since lambda functions, just like any function, has a value, we can call it immediately:

```
scm> ((lambda (x) (* x 2)) 5)
10
```

We can even assign a name to the function, making it seem like a define procedure.

```
scm> (define doubly (lambda (x) (* x 2)))
doubly
scm> (doubly 5)
10
```

Let Expressions

There is another form similar to a lambda expression called a Let expression.

Here, we define variables from the start, and then run the block of code using those variable assignments.

Let's look at an example:

```
scm> (let ((x 5) (y 6)) (* x y))
30
```

We have two cases: the defining expression, and the body expression. In the defining expression, we assign 5 to the variable x and 6 to the variable y. These variables are only used inside the let frame. The body, in this case, multiplies the two variables defined.

Practice

Define a function that takes in a number and returns a function that multiplies any input by that number.

```
(define (multiply-out x)



        ____________________________________
)
;((multiply-out 5) 6)
;30
```

Factorial function

```
(define (factorial x)









)
;(factorial 5)
;120
;(factorial 1)
;1
```

Lists

The process of making a list in Scheme is the same idea as a linked list. The syntax is slightly different:

cons: Constructing a list. Cons takes in two arguments, the first can be anything, the second can either be another list, or null. (Similar to a Linked List from Python)

car: The FIRST item in the first linked list pair.

cdr: The REST item in the first linked list pair.

Tip: In Scheme, we use 'nil' as null or an empty list.

Let's look at some examples:

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

Here, we made a list with one item.

```
scm> (cons 2 (cons 3 nil))
(2 3)
```

Here, we made a list with two items. The second argument for the first cons is another list.

```
scm> (car (cons 2 (cons 3 nil)))
2
scm> (cdr (cons 2 (cons 3 nil)))
(3)
```

By applying car to the list (2 3), we are able to take the first value.

By applying cdr to the list, we can take the rest value, which is a LIST with one value (3), not the actual number 3.

If the second argument in cons happens to not be a list or nil, we create a **malformed list**.

Let's look at some examples:

```
scm> (cons 6 (cons 1 nil))
(6 1)
scm> (cons 6 1)
(6 . 1)
scm> (cdr (cons 6 1))
1
scm> (cdr (cons 6 (cons 1 nil)))
(1)
```

If the list is malformed, there is a dot in between the pair of values. The dot is not there if the list is well-formed.

Extraneous Syntax

```
scm> (list 6 1 9)
(6 1 9)
scm> '(list 6 1 9)
(list 6 1 9)
scm> '(6 1 9)
```

```
(6 1 9)
```

These are different ways to make a list of values. Notice the second example, even though the list procedure is inside the parenthesis, Scheme assumes it is a string because of the quote on the outside.

```
scm> 'string
string
scm> (eq? 'this 'this)
#t
scm> (equal? 'this 'this)
#t
scm> (equal? 'that 'this)
#f
scm> (eq? '(1 2 3) '(1 2 3))
#f
scm> (equal? '(1 2 3) '(1 2 3))
#t
```

eq? works like == for primitives, but for things like lists, works like *is*.

equal? compares the value itself.

```
scm> (even? 2)
#t
scm> (odd? 2)
#f
scm> (list? '(2))
#t
```

Practice

Draw the box and pointer diagram for the following lists and write out the final result.

```
scm> (list 10 (list 1 2 (cons 1 2) (cons 3 (cons 4 (list 'list 1))) 'hi 10)
(cons 10 nil))
```

```
scm> (cons 3 (list 10 (or (cons 10 (list 9 2)) 0) 3 (cons 3 3) 3))
```

```
scm> (list 'denero 'denero (or 'denero) '(denero nil denero) (cons 'denero
(cons 'denero (list 'denero (list 'd 'e 'n 'e (cons 'r 'o))))))
```

Double each value in a given list:

```
(define (deeply-doubled lst)









  )
;(deeply-doubled '(2 3 4 5 6))
;(4 6 8 10 12)
;(deeply-doubled (list 2 3 (list 34 5) 5 6))
;(4 6 (68 10) 10 12)
```