In the next part of the course, we will be working with the Scheme programming language. In addition to learning how to write Scheme programs, we will eventually write a Scheme interpreter in Project 4!

Scheme is a dialect of the Lisp programming language, a language dating back to 1958. The popularity of Scheme within the programming language community stems from its simplicity – in fact, previous versions of CS 61A were taught in the Scheme language.

1 The Scheme Interpreter

Like Python, the Scheme language has an interactive interpreter. On your class accounts, you can access this by typing in stk.

```
nova [416] ~ # stk
STk>
```

Keep this one thing in mind: **everything is an expression.**
Primitives

Scheme has a certain set of atomic primitive expressions. Atomic means that these expressions cannot be divided up, or at least are not usually created out of smaller expressions. The ones we’ll use the most are numbers (i.e., integers and floats), the two boolean values, and symbols. Scheme also has strings, but we won’t be using them too often.

```
scm> 123
123
scm> 123.123
123.123
scm> #t
#t
scm> #f
#f
scm> "asdf"
"asdf"
scm> 'a ; this is a symbol
a
```

You’ll notice that numbers work pretty much the same in Scheme as they do in Python. Also, instead of True and False, we have #t and #f. Before explaining symbols, though, let’s look at how we define variables:

```
scm> (define a 3)
a
scm> a
3
```

You’ll notice this is different. In Scheme, you need to have define, followed by the name of the variable, followed by the expression (which first gets evaluated) that you want bound to the name. The space between each of the parts (except the parentheses) matters. Also, notice that to refer back to a, we just type in a, as in Python.

An important thing to note is that a is a symbol. When you type things into the interpreter, Scheme will evaluate them. The rule for evaluating a symbol is to get the value bound to that symbol. This is one difference between strings and symbols – symbols don’t evaluate to themselves. However, as you saw above, when you type in ‘a, you get a. This is because when you use the single quote, you’re telling Scheme not to follow the normal rules of evaluation and just have the symbol return as itself. You’ll also notice that you can use the single quotes on integers, floats, and booleans. However they’re unnecessary because those evaluate to themselves. Finally, let’s revisit what you wrote when you did (define a 3). Here, both define and a are symbols!
2.1 Questions

1. What would Scheme print?
   
   scm> (define a 1)
   a
   scm> (define b a)
   b
   scm> (define c 'a)
   c


3 Evaluating Function Calls and Special Forms

Now, just defining variables and printing out primitives isn’t very useful. You want to call functions too:

   scm> (+ 1 2)
   3
   scm> (- 2 3)
   -1
   scm> (* 6 3)
   18
   scm> (/ 5 2)
   2.5
   scm> (+ 1 (* 3 4))
   13

3.1 Functions

You might notice that Scheme represents function calls differently. To call a function in Scheme, you give the symbol for the function name, then you give the arguments (remember the spaces!). Evaluating a Scheme function call works just like Python: first, evaluate the operator (the first expression after the ()), then evaluate each of the arguments. Then, apply the operator to those evaluated arguments. So when you evaluate (+ 1 2), you evaluate the + symbol which is bound to a built-in addition function, then you evaluate 1 and 2. Finally, you apply the addition function to 1 and 2.
Some important functions you’ll want to use are:

- +, -, *, /
- eq?, =, >, >=, <, <=

### 3.2 Questions

1. What do the following return?
   - (+ 1)
   - (* 3)
   - (+ (* 3 3) (* 4 4))
   - (define a (define b 3))
   - a

### 3.3 Special Forms

However, there are certain things that look like function calls that don’t follow this rule for evaluation. These are called special forms. You’ve already seen one – `define`, where the first argument, the symbol “define,” doesn’t actually get evaluated to a value. Another one we’ll use a lot is `if`.

An `if` expression looks like:

```
(if <CONDITION> <THEN> <ELSE>)
```

where `<CONDITION>`, `<THEN>` and `<ELSE>` are expressions. Here’s how it gets evaluated. First, `<CONDITION>` is evaluated. If it evaluates to `#f`, then `<ELSE>` is evaluated. Otherwise, `<THEN>` is evaluated. **Every atomic expression that is not `#f` evaluates to “true.”**

```
scm> (if ’this-evaluates-to-true 1 2)
1
s scm> (if #f (/ 1 0) ’this-gets-returned)
this-gets-returned
```

There are also special forms for the boolean operators which exhibit the same short-circuiting behavior that you see in Python. The return values are either the value that lets you know the expression evaluates to a true value or `#f`. 
3.4 Questions

1. What does the following do?

```
scm> (if (or #t (/ 1 0)) 1 (/ 1 0))
```

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

```
scm> ((if (< 4 3) + -) 4 100)
```

4 Lambda, Environments and Defining Functions

Scheme has lambdas too! In fact, lambdas are more powerful in Scheme than in Python. The syntax is `(lambda (PARAMETERS) <EXPR>)`. Like in Python, lambdas are function values. Also like in Python, when a lambda expression is called in Scheme, a new frame is created where the symbols defined in the <PARAMETERS> section are bound to the arguments passed in. Then, <EXPR> is evaluated under this new frame. Note that <EXPR> is not evaluated until the lambda function is called.

```
scm> (define x 3)
x
scm> (define y 4)
y
scm> ((lambda (x y) (+ x y)) 6 7)
13
```
Like in Python, lambda functions are also values! So you can do this to define functions:

```scheme
(scm> (define square (lambda (x) (* x x)))
square
(scm> (square 4)
16
```

This can be a bit tedious though. Luckily Scheme has a shortcut – our old friend define:

```scheme
(scm> (define (square x) (* x x))
square
(scm> (square 5)
25
```

When you do `(define (<FUNCTION NAME> <PARAMETERS>) <EXPR>)`, Scheme will automatically transform it to `(define <FUNCTION NAME> (lambda (<PARAMETERS>) <EXPR>)`. In this way, lambdas are more central to Scheme than they are to Python.

There is also special form based around `lambda`: let. The structure of `let` is as follows:

```scheme
(let ( (<SYMBOL1> <EXPR1>)
      ...
      (<SYMBOLN> <EXPRN>))
     <BODY>)
```

This special form really just gets transformed to:

```scheme
( (lambda (<SYMBOL1> ... <SYMBOLN>) <BODY>) <EXPR1> ... <EXPRN>)
```

Essentially, let just binds symbols to expressions, then runs its body. This can be useful if you need to reuse a value multiple times, or if you want to make your code more readable:

```scheme
(define (sin x)
  (if (< x 0.000001)
      x
      (let ( (recursive-step (sin (/ x 3))) )
         (- (* 3 recursive-step)
             (* 4 (expt recursive-step 3))))))
```

## 4.1 Questions

1. Write a function that calculates factorial. (Note we have not seen any iteration yet.)

```scheme
(define (factorial x) )
```
2. Write a function that calculates the $n^{th}$ Fibonacci number.

\[
\text{(define (fib n)} \n\text{  (if (< n 2)} \n\text{    1)} \n\text{  )})
\]

5 Pairs and Lists

So far, we have lambdas and a few atomic primitives. How do we create larger, more complicated data structures? Well, the most important data structure in Scheme is the pair. A pair is an abstract data type with the constructor cons (which takes two arguments), and two selectors, car and cdr (which get the first and second argument respectively). car and cdr don’t stand for anything anymore, but if you want the history go to "http://en.wikipedia.org/wiki/CAR_and_CDR"

```
scm> (define a (cons 1 2))
a
scm> a
(1 . 2)
scm> (car a)
1
scm> (cdr a)
2
```

Note that when a pair is printed, the car and cdr elements are separated by a period. Remember, cons always takes in exactly two arguments.

A common data structure that you build out of pairs is the list. A list is either the empty list, which is another primitive represented as ’() or nil, or a cons pair where the cdr is a list. (Note the similarity to Links!)

```
scm> ’()
()
scm> nil
()
scm> (cons 1 (cons 2 nil))
(1 2)
scm> (cons 1 (cons 2 (cons 3 nil)))
(1 2 3)
```
Note that there are no dots here. When a dot is followed by a left parenthesis, the dot, left parenthesis, and matching right parenthesis are deleted. You can check if a list is nil with the null? function.

A shorthand for writing out a list is:

```scheme
scm> '(1 2 3)
(1 2 3)
scm> '(define (square x) (* x x))
(define (square x) (* x x))
```

You might notice that the evaluation of the second expression looks a lot like Scheme code. That’s because Scheme code is made up of lists. When you quote an expression (like a list), you’re telling Scheme not to evaluate the expression, but instead keep it as is. This is one of the reasons why Scheme is cool – it can be defined within itself!

### 5.1 Questions

1. Fill in the following to complete an abstract data type for binary trees, in which each node has at most 2 children, left and right:

   ```scheme
   (define (make-btree entry left right)
     (cons entry (cons left right)))
   (define (entry tree))
   (define (left tree))
   (define (right tree))
   ```

2. Using the above definition, write a function that sums up the entries of a binary tree, assuming that the entries are all numbers.

   ```scheme
   (define (btree-sum tree))
   ```
3. Define map, where the first argument is a function and the second a list. This should work like Python’s map.
   
   (define (map fn lst)

   )

4. Define reduce, where the first argument is a function that takes two arguments, the second is a starting value, and the third is a list. This should work like Python’s reduce.
   
   (define (reduce fn s lst)

   )

5.2 Extra Questions

1. Write a Scheme function that, when given an element, a list, and a position, inserts the element into the list at that position.
   
   (define (insert element lst position)

   )

2. Write a Scheme function that, when given a list, such as (1 2 3 4), duplicates every element in the list (i.e. (1 1 2 2 3 3 4 4)).
   
   (define (duplicate lst)

   )