1 Introduction

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.

2 Primitives

Scheme has a set of atomic primitive expressions. Atomic means that these expressions cannot be divided up.

```
 scm> 123
 123
 scm> 123.123
 123.123
 scm> #t
 True
 scm> #f
 False
```

Unlike in Python, the only primitive in Scheme that is a false value is #f and its equivalents, false and False. The define special form defines variables and procedures
by binding a value to a variable, just like the assignment statement in Python. When a variable is defined, the define special form returns a symbol of its name. A procedure is what we call a function in Scheme!

The syntax to define a variable and procedure are:

- (define <variable name> <value>)
- (define (<function name> <parameters>)<function body>)

```scheme
scm> (define a 3) ; a = 3
a
scm> a
3
scm> (define (foo x) x) ; procedure named foo
foo
scm> (foo a)
3
```

### 2.1 Questions

1. What would Scheme print?
   ```scheme
   scm> (define a 1)
   scm> a
   scm> (define b a)
   scm> b
   scm> (define c 'a)
   scm> c
   ```
3 Call Expressions

Scheme call expressions follow prefix notation, where an operator is followed by zero or more operand subexpressions. Operators may be symbols, such as + and * or more complex expressions, as long as they evaluate to procedure values.

```
scm> (- 1 1) ; 1 - 1
0
scm> (/ 8 4 2) ; 8 / 4 / 2
1
scm> (* (+ 1 2) (+ 1 2)) ; (1 + 2) * (1 + 2)
9
```

To call a function in Scheme, you first need a set of parentheses. Inside the parentheses, you specify a function, then the arguments (remember the spaces!).

Evaluating a Scheme function call works just like Python:

1. Evaluate the operator (the first expression after the ( ), then evaluate each of the operands.
2. Apply the operator to those evaluated operands.

When you evaluate \((+ 1 2)\), you evaluate the + symbol, which is bound to a built-in addition function. Then, you evaluate 1 and 2, which are primitives. Finally, you apply the addition function to 1 and 2.

Some important built-in functions you’ll want to know are:

- \(+, -, *, /\)
- \(equal?, =, >, >=, <, <=\)

3.1 Questions

1. What would Scheme print?
   
   ```
   scm> (+ 1)
   scm> (* 3)
   scm> (+ (* 3 3) (* 4 4))
   scm> (define a (define b 3))
   scm> a
   scm> b
   ```
There are certain expressions that look like function calls, but don’t follow the rule for order of evaluation. These are called special forms. You’ve already seen one — define, where the first argument, the variable name, doesn’t actually get evaluated to a value.

4.1 If Statements

Another common special form is the if form. An if expression looks like:

\[
\text{if} \ <\text{condition}> \ <\text{then}> \ <\text{else}>
\]

where <condition>,<then> and <else> are expressions. First, <condition> is evaluated. If it evaluates to #t, then <then> is evaluated. Otherwise, <else> is evaluated. Remember that only False and #f are false-y values; everything else is truth-y.

scm> (if (< 4 5) 1 2) 1
scm> (if #f (/ 1 0) 42) 42

4.2 Boolean Operators

Scheme also has boolean operators and, or, and not like in Python! In addition, and and or are also special forms because they are short-circuiting operators.

scm> (and 1 2 3) 3
scm> (or 1 2 3) 1
scm> (or True (/ 1 0)) True
scm> (and False (/ 1 0)) False
scm> (not 3) False
scm> (not True) False
4.3 Questions

1. What does Scheme print?
   
   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)
   
   scm> (if 0 1 2)

4.4 Lambdas and Defining Functions

Scheme has lambdas too! 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 parameters are bound to the arguments passed in. Then, <EXPR> is evaluated in 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:

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

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.
4.5 Let

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

\[
\text{let} \ ( (\text{<SYMBOL1>} \ \text{<EXPR1>}) \ \\
\ldots \ \\
(\text{<SYMBOLN>} \ \text{<EXPRN>}) ) \ \\
\text{<BODY>} 
\]

This special form is really just equivalent to:

\[
( (\lambda (\text{<SYMBOL1>} \ \ldots \ \text{<SYMBOLN>}) \ \text{<BODY>}) \ \text{<EXPR1>} \ \ldots \ \text{<EXPRN>})
\]

let effectively binds symbols to expressions, then runs the body of the let form. This can be useful if you need to reuse a value multiple times, or if you want to make your code more readable.

For example, we can use the approximation \( \sin(x) \approx x \) (which is true for small \( x \)) and the trigonometric identity \( \sin(x) = 3 \sin(x/3) - 4 \sin^3(x/3) \) to approximate \( \sin(x) \) for any \( x \).

\[
\text{(define (sin x)} \\
\text{\quad (if (< x 0.000001) \text{x}} \\
\text{\quad \quad (let ( (recursive-step (sin (/ x 3))) )} \\
\text{\quad \quad \quad (- (* 3 recursive-step) \text{))})})
\]

4.6 Questions

1. Write a function that calculates factorial. (Note we have not seen any iteration yet.)
   \[
   \text{(define (factorial x)} \\
   \text{)}
   \]

2. Write a function that calculates the \( n^{th} \) Fibonacci number.
   \[
   \text{(define (fib n)} \\
   \text{\quad (if (< n 2) \text{1}} \\
   \text{\quad \quad 1)}
   \text{)}
   \]
5 Pairs and Lists

To construct a (linked) list in Scheme, you can use the constructor \texttt{cons} (which takes two arguments). \texttt{nil} represents the empty list. If you have a linked list in Scheme, you can use selector \texttt{car} to get the first element and selector \texttt{cdr} to get the rest of the list. (\texttt{car} and \texttt{cdr} don’t stand for anything anymore, but if you want the history go to \url{http://en.wikipedia.org/wiki/CAR_and_CDR}.)

\begin{verbatim}
scm> nil
()
scm> (null? nil)
#t
scm> (cons 2 nil)
(2)
scm> (cons 3 (cons 2 nil))
(3 2)
scm> (define a (cons 3 (cons 2 nil)))
a
scm> (car a)
3
scm> (cdr a)
(2)
scm> (car (cdr a))
2
scm> (define (len a)
    (if (null? a)
        0
        (+ 1 (len (cdr a)))))
len
scm> (len a)
2
\end{verbatim}
If a list is a "good looking" list, like the ones above where the second element is always a linked list, we call it a **well-formed list**. Interestingly, in Scheme, the second element does not have to be a linked list. You can give something else instead, but `cons` always takes exactly 2 arguments. These lists are called **malformed list**. The difference is a dot:

```scheme
scm> (cons 2 3)
(2 . 3)
scm> (cons 2 (cons 3 nil))
(2 3)
scm> (cdr (cons 2 3))
3
scm> (cdr (cons 2 (cons 3 nil)))
(3)
```

In general, the rule for displaying a pair is as follows: use the dot to separate the **car** and **cdr** fields of a pair, but if the dot is immediately followed by an open parenthesis, then remove the dot and the parenthesis pair. Thus, `(0 . (1 . 2))` becomes `(0 1 . 2)`

There are many useful operations and shorthands on lists. One of them is **list special form**. `list` takes zero or more arguments and returns a list of its arguments. Each argument is in the car field of each list element. It behaves the same as quoting a list, which also creates the list.

```scheme
scm> (list 1 2 3)
(1 2 3)
scm> '(1 2 3)
(1 2 3)
scm> (car '(1 2 3))
1
scm> (equal? '(1 2 3) (list 1 2 3))
#t
scm> '(1 . (2 3))
(1 2 3)
scm> '(define (square x) (* x x))
(define (square x) (* x x))
```
1. Define a function that takes 2 lists and concatenates them together. Notice that simply calling `(cons a b)` would not work because it will create a deep list. Instead, think recursively!

```scheme
(define (concat a b)
  ...)
```

```scheme
scm> (concat '(1 2 3) '(2 3 4))
(1 2 3 2 3 4)
```

2. Define `replicate`, which takes an element `x` and a non-negative integer `n`, and returns a list with `x` repeated `n` times.

```scheme
(define (replicate x n)
  ...)
```

```scheme
scm> (replicate 5 3)
(5 5 5)
```
3. A **run-length encoding** is a method of compressing a sequence of letters. The list \((a \ a \ a \ b \ a \ a \ a \ a)\) can be compressed to \(((a \ 3) \ (b \ 1) \ (a \ 4))\), where the compressed version of the sequence keeps track of how many letters appear consecutively.

Write a Scheme function that takes a compressed sequence and expands it into the original sequence. *Hint:* try to use functions you defined earlier in this worksheet.

```
(define (uncompress s)
  ...)
```

```
scm> (uncompress '((a 1) (b 2) (c 3)))
(a b b c c c)
```
4. Define deep-apply, which takes a nested list and applies a given procedure to every element. deep-apply should return a nested list with the same structure as the input list, but with each element replaced by the result of applying the given procedure to that element. Use the built-in `list?` procedure to detect whether a value is a list. The procedure `map` has been defined for you.

\[
(\text{define} (\text{map} \ fn \ lst))
\]

\[
(\text{if} (\text{null?} \ lst)
  \text{nil}
  (\text{cons} (fn \ (\text{car} \ lst)) (\text{map} \ fn \ (\text{cdr} \ lst)))))
\]

\[
(\text{define} \ (\text{deep-apply} \ fn \ \text{nested-list})
\)

\[
\text{scm> } (\text{deep-apply} \ (\lambda (x) (* x x)) \ '(1 2 3))
\]

\[
(1 4 9)
\]

\[
\text{scm> } (\text{deep-apply} \ (\lambda (x) (* x x)) \ '(1 ((4) 5) 9))
\]

\[
(1 ((16) 25) 81)
\]

\[
\text{scm> } (\text{deep-apply} \ (\lambda (x) (* x x)) \ 2)
\]

\[
4
\]
Extra Questions

1. Fill in the following to complete an abstract tree data type:
   \[
   \text{(define (make-tree root branches) (cons root branches))}
   \]

   \[
   \text{(define (root tree) )}
   \]

   \[
   \text{(define (branches tree) )}
   \]

2. Using the abstract data type above, write a function that sums up the entries of a tree, assuming that the entries are all numbers. Hint: you may want to use the `map` function you defined above, as well as an additional helper function.
   \[
   \text{(define (tree-sum tree) )}
   \]

3. Using the abstract data type above, write a Scheme function that creates a new tree where the entries are the product of the entries along the path to the root in the original tree. Hint: you may want to write helper functions.
   \[
   \text{(define (path-product-tree t) )}
   \]