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>)

```
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?
   ```
   scm> (define a 1)
   
   scm> a
   
   scm> (define b a)
   
   scm> b
   
   scm> (define c 'a)
   
   scm> c
   ```
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.

```scheme
scent> (- 1 1) ; 1 - 1
0
scent> (/ 8 4 2) ; 8 / 4 / 2
1
scent> (* (+ 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?

```scheme
scent> (+ 1)

scent> (* 3)

scent> (+ (* 3 3) (* 4 4))

scent> (define a (define b 3))

scent> a
scent> b
```
4 Special Forms

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:

```
(if <condition> <then> <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 \texttt{lambda}: \texttt{let}. The structure of \texttt{let} is as follows:
\begin{verbatim}
(let ( (<SYMBOL1> <EXPR1>)
... 
(<SYMBOLN> <EXPRN>) )
<BODY> )
\end{verbatim}

This special form is really just equivalent to:
\begin{verbatim}
( (lambda ( <SYMBOL1> ... <SYMBOLN>) <BODY>) <EXPR1> ... <EXPRN>)
\end{verbatim}
\texttt{let} effectively binds symbols to expressions, then runs the body of the \texttt{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 $x$.
\begin{verbatim}
(define (sin x)
  (if (< x 0.000001)
    x
    (let ( (recursive-step (sin (/ x 3))) )
       (- (* 3 recursive-step)
          (* 4 (expt recursive-step 3)))))
\end{verbatim}

4.6 Questions

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

)
\end{verbatim}

2. Write a function that calculates the $n^{th}$ Fibonacci number.
\begin{verbatim}
(define (fib n)
  (if (< n 2)
    1

)
\end{verbatim}
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}.)

```
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
```
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 \texttt{cons} always takes exactly 2 arguments. These lists are called malformed list. The difference is a dot:

\texttt{srm> (cons 2 3)}
\begin{verbatim}
(2 . 3)
\end{verbatim}
\texttt{srm> (cons 2 (cons 3 nil))}
\begin{verbatim}
(2 3)
\end{verbatim}
\texttt{srm> (cdr (cons 2 3))}
\begin{verbatim}
3
\end{verbatim}
\texttt{srm> (cdr (cons 2 (cons 3 nil)))}
\begin{verbatim}
(3)
\end{verbatim}

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, \texttt{(0 . (1 . 2))} becomes \texttt{(0 1 . 2)}

There are many useful operations and shorthands on lists. One of them is \texttt{list} special form \texttt{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.

\texttt{srm> (list 1 2 3)}
\begin{verbatim}
(1 2 3)
\end{verbatim}
\texttt{srm> '(1 2 3)}
\begin{verbatim}
(1 2 3)
\end{verbatim}
\texttt{srm> (car '(1 2 3))}
\begin{verbatim}
1
\end{verbatim}
\texttt{srm> (equal? '(1 2 3) (list 1 2 3))}
\begin{verbatim}
#t
\end{verbatim}
\texttt{srm> '(1 . (2 3))}
\begin{verbatim}
(1 2 3)
\end{verbatim}
\texttt{srm> '(define (square x) (* x x))}
\begin{verbatim}
(define (square x) (* x x))
\end{verbatim}
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!

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

   ```scm```
   ```scm> (concat '(1 2 3) '(2 3 4))```
   `(1 2 3 2 3 4)`
   ```scm> (concat '((1 2) (3 4)) '(5 6))```
   `((1 2) (3 4) 5 6)`

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

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

   ```scm```
   ```scm> (replicate 5 3)```
   `(5 5 5)`

   ```scm> (replicate '() 1)```
   `()`
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)
  ...
  (a b b c c c)
```

```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.

```scheme
(define (map fn lst)
  (if (null? lst)
      nil
      (cons (fn (car lst)) (map fn (cdr lst))))))

(define (deep-apply fn nested-list)
)
```

```
scm> (deep-apply (lambda (x) (* x x)) '(1 2 3))
(1 4 9)
scm> (deep-apply (lambda (x) (* x x)) '(1 ((4) 5) 9))
(1 ((16) 25) 81)
scm> (deep-apply (lambda (x) (* x x)) 2)
4
```
1. Fill in the following to complete an abstract tree data type:

\[
\begin{align*}
\text{(define (make-tree root branches) (cons root branches))} \\
\text{(define (root tree) )} \\
\text{(define (branches tree) )}
\end{align*}
\]

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.

\[
\begin{align*}
\text{(define (tree-sum tree) )}
\end{align*}
\]

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.

\[
\begin{align*}
\text{(define (path-product-tree t) )}
\end{align*}
\]