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>)`
Questions

2.1 What would Scheme display?

```scheme
(scm> (define a 1))

s<cm> a

(scm> (define b a))

s<cm> b

(scm> (define c 'a))

s<cm> c
```

3 Call Expressions

To call a function in Scheme, you first need a set of parentheses. Inside the parentheses, you specify an operator expression, then zero or more operand subexpressions (remember the spaces!).

Operators may be symbols, such as + and * or more complex expressions, as long as they evaluate to procedure values.

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

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.
Questions

3.1 What would Scheme display?

```scheme
scm> (+ 1)
```

```scheme
scm> (* 3)
```

```scheme
scm> (+ (* 3 3) (* 4 4))
```

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

```scheme
scm> a
```

```scheme
scm> 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 Expression

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

```scheme
(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 #f is a false-y value (also False for our interpreter); everything else is truth-y.

```scheme
scm> (if (< 4 5) 1 2)
1
scm> (if #f (/ 1 0) 42)
42
```
4.2 Boolean Operators

Much like Python, Scheme also has the boolean operators \texttt{and}, \texttt{or}, and \texttt{not}. In addition, \texttt{and} and \texttt{or} are also special forms because they are short-circuiting operators.

\begin{verbatim}
scm> (and 25 32)
32
scm> (or 1 2)
1
\end{verbatim}

Questions

4.1 What would Scheme display?

\begin{verbatim}
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)
\end{verbatim}

4.3 Lambdas and Defining Functions

Scheme has lambdas too! The syntax is

\begin{verbatim}
(lambda (<PARAMETERS>) <EXPR>)
\end{verbatim}

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, \texttt{EXPR} is evaluated in this new frame. Note that \texttt{EXPR} is not evaluated until the lambda function is called.

Like in Python, lambda functions are also values! So you can do this to define functions:

\begin{verbatim}
scm> (define (square x) (* x x)) ; Create function square using define special form
square
scm> (define square (lambda (x) (* x x))) ; Equivalently, bind the name square to a lambda function
square
scm> (square 4)
16
\end{verbatim}
let is another special form based around lambda. The structure of let is as follows:

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

This binds the results of evaluating expressions 1 through n to their associated symbols, creating temporary variables. Finally, the body of the let is evaluated.

This special form is really just equivalent to:

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

Think of the temporary variables as being the parameters of a lambda function. Then, the arguments are the values of the expressions, which we bind to the temporary variables by calling the lambda.

Consider the following example:

(let ( (x 1)
      (y 2))
    (+ x y))

This is equivalent to:

((lambda (x y) (+ x y)) 1 2)

Questions

4.1 Write a function that calculates the factorial of a number.

(define (factorial x)
  
)

4.2 Write a function that calculates the \( n^{th} \) Fibonacci number.

(define (fib n)
  (if (< n 2)
      1
      )))
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 \textbf{well-formed list}. Interestingly, in Scheme, the second element does not have to be a linked list. You can supply something else instead, creating a \textbf{malformed list}. The difference is shown with a dot:

\begin{verbatim}
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)
\end{verbatim}

In general, the rule for displaying a pair is as follows: use the dot to separate the \texttt{car} and \texttt{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, \((\emptyset \ . \ (1 \ . \ 2))\) becomes \((\emptyset \ 1 \ . \ 2)\)

There are many useful operations and shorthands on lists. \texttt{list} takes zero or more arguments and returns a list of its arguments.

This typically behaves much like quoting a list, except that quoting will not evaluate the list you have quoted which can result in some different outcomes.

\begin{verbatim}
> (list 1 2 3)
(1 2 3)
> '(1 2 3)
(1 2 3)
> (car '(1 2 3))
1
> (equal? '(1 2 3) (list 1 2 3))
#t
> '(1 . (2 3))
(1 2 3)
> (define (square x) (* x x))
(define (square x) (* x x))
> square ; We didn't actually define square above because of the quote
Error
> (list (cons 1 2))
((1 . 2))
> '((cons 1 2))
((cons 1 2))

=, eq?, equal?

- = can only be used for comparing numbers.

- eq? behaves like == in Python for comparing two non-pairs (numbers, booleans, etc.). Otherwise, eq? behaves like is in Python.

- equal? compares pairs by determining if their cars are equal? and their cdrs are equal?(that is, they have the same contents). Otherwise, equal? behaves like eq?.

\begin{verbatim}
> (define a '(1 2 3))
a
> (= a a)
Error
> (equal? a '(1 2 3))
#t
> (eq? a '(1 2 3))
#f
> (define b a)
b
> (eq? a b)
#t
\end{verbatim}
Questions

5.1 Define \texttt{concat}, which takes two lists and concatenates them.

Notice that simply calling \texttt{(cons a b)} would not work because it will create a deep list.

\begin{verbatim}
(define (concat a b) )
\end{verbatim}

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

5.2 Define \texttt{replicate}, which takes an element \texttt{x} and a non-negative integer \texttt{n}, and returns a list with \texttt{x} repeated \texttt{n} times.

\begin{verbatim}
(define (replicate x n) )
\end{verbatim}

scm> (replicate 5 3)
(5 5 5)
5.3 A run-length encoding is a method of compressing a sequence of letters. The list \((a\ a\ a\ b\ a\ a\ a)\) 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. \textit{Hint:} You may want to use \texttt{concat} and \texttt{replicate}.

\begin{verbatim}
(define (uncompress s)
  )

scm> (uncompress '((a 1) (b 2) (c 3)))
  (a b b c c c)
\end{verbatim}

5.4 Define \textbf{map}, which takes a procedure and applies it to every element in a given list.

\begin{verbatim}
(define (map fn lst)
  )

scm> (map (lambda (x) (* x x)) '(1 2 3))
  (1 4 9)
\end{verbatim}

5.5 Define \textbf{deep-map}, which takes a procedure and applies it to every element in a given nested list.

The result should be a nested list with the same structure as the input list, but with each element replaced by the result of applying the procedure to that element.

Use the built-in \texttt{list?} procedure to detect whether a value is a list.

\begin{verbatim}
(define (deep-map fn lst)
  )

scm> (deep-map (lambda (x) (* x x)) '(1 2 3))
  (1 4 9)
scm> (deep-map (lambda (x) (* x x)) '(1 ((4) 5) 9))
  (1 ((16) 25) 81)
\end{verbatim}
6 Extra Questions

6.1 Fill in the following to complete an abstract tree data type:

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

6.2 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.

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

6.3 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.

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