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.

```scheme
scl> 123
123
scl> 123.123
123.123
scl> #t
True
scl> #f
False
scl> 'a ; this is a symbol
a
```
To define variables:

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

```
a
```

```
scm> a
```

```
3
```

The `define` statement binds a value to a variable (just like the assignment operator in Python); in addition, define returns the variable name (in this case, `a`).

More precisely, `define` returns the symbol `a`. As you saw above, when you type `'a`, you also get the symbol `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.

### 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

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

```scheme
scm> (+ 1 2)
3
scm> (/ 5 2)
2.5
scm> (+ 1 (* 3 4))
13
```

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:

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

3.1 Questions

1. What would Scheme print?

```scheme
scm> (+ 1)
scm> (* 3)
scm> (+ (* 3 3) (* 4 4))
scm> (define a (define b 3))
scm> a
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 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 False, then <ELSE> is evaluated. Otherwise, <THEN> is evaluated. Only False and #f evaluate to False; everything else is truth-y.

\[
\text{scm} > (\text{if } (< 4 5) 1 2)
\]

1

\[
\text{scm} > (\text{if } \text{False} (/ 1 0) 42)
\]

42

4.2 Boolean operators

Boolean operators (and and or) are also special forms because they are short-circuiting operators (just like in Python).

\[
\text{scm} > (\text{and } 1 2 3)
\]

3

\[
\text{scm} > (\text{or } 1 2 3)
\]

1

\[
\text{scm} > (\text{or } \text{True} (/ 1 0))
\]

True

\[
\text{scm} > (\text{and } \text{False} (/1 0))
\]

False

\[
\text{scm} > (\text{not } 3)
\]

False

\[
\text{scm} > (\text{not } \text{True})
\]

False
### 4.3 Questions

1. What does Scheme print?

   ```scheme
   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.

```scheme
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.
4.5 Let

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

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

This special form really just gets transformed to:

```
( (lambda (<SYMBOL1> ... <SYMBOLN>) <BODY>) <EXPR1> ... <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 \( x \).

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

4.6 Questions

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

```
(define (factorial x)
)
```

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

```
(define (fib n)
   (if (< n 2)
      1
      )
)
```
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 \texttt{cons} (which takes two arguments), and two selectors, \texttt{car} and \texttt{cdr} (which get the first and second argument respectively). \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> (define a (cons 1 2))
a
scm> a
(1 . 2)
scm> (car a)
1
scm> (cdr a)
2
\end{verbatim}

Note that when a pair is printed, the \texttt{car} and \texttt{cdr} elements are separated by a period. Remember, \texttt{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 \texttt{nil}, or a \texttt{cons} pair where the \texttt{cdr} is a list. (Note the similarity to Links!)

\begin{verbatim}
scm> '()
()
scm> nil
()
scm> (cons 1 (cons 2 nil))
(1 2)
\end{verbatim}

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 \texttt{nil} with the \texttt{null?} function.

A shorthand for writing out a list is:

\begin{verbatim}
scm> '(1 2 3)
(1 2 3)
scm> '(define (square x) (* x x))
(define (square x) (* x x))
\end{verbatim}

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. Define `map`, which takes function `fn` and a list `lst`. It applies the function `fn` to every element of `lst` and returns a list containing the results.

   ```scheme
   (define (map fn lst)
     ...)
   
   scm> (map (lambda (x) (* x x)) '(1 2 3))
   (1 4 9)
   ```

2. Define `concat`, which takes a list of lists, and constructs a list by concatenating all the elements together into one list. Use the built-in `append` function to concatenate two lists.

   ```scheme
   (define (concat lsts)
     ...)
   
   scm> (append '(1 4 7) '(2 5 8))
   (1 4 7 2 5 8)
   scm> (concat '(((1 4 7) (2 5 8) (3 6 9)))
   (1 4 7 2 5 8 3 6 9)
   ```
3. Define \texttt{replicate}, which takes an element \textit{x} and a non-negative integer \textit{n}, and returns a list with \textit{x} repeated \textit{n} times.

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

\texttt{scm> (replicate 5 3)}
\texttt{(5 5 5)}

4. A \textbf{run-length encoding} is a method of compressing a sequence of letters. The list \texttt{(a a a b a a a a)} can be compressed to \texttt{((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:} try to use functions you defined earlier in this worksheet.

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

\texttt{scm> (uncompress '((a 1) (b 2) (c 3)))}
\texttt{(a b b c c c)}
5. Define `deep-apply`, which takes a nested list and applies a given function to every element. For the purposes of this question, a *nested list* is either

- a single element (e.g. 4)
- a list of nested lists (e.g. (1 ((4) 5) 9)).

`deep-apply` should return a nested list with the same shape as the input list, but with each element replaced by the result of applying the given function to that element. Use the built-in `list?` function to detect whether a value is a list.

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

```scheme
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:
   `(define (make-tree entry children) (cons entry children))`

   `(define (entry tree) ...)`

   `(define (children 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.
   `(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.

```
(define (path-product-tree t)
  ; your implementation here
)
```