CS 61A
Summer 2020

Scheme, Scheme Lists

Discussion 10: July 28, 2020

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 and Defining Variables

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

```
scm> 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`. **This means that 0 is not 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>)`

Special forms are types of expressions with unique evaluation rules that can do a variety of things. Often times, speical forms are analagous to statements in Python, such as assignment statements, `if` statements, and `def` statements. However, all special forms in Scheme evaluate to a value. We'll learn more about special forms later in the discussion.

3 Call Expressions

Call expressions apply a procedure to some arguments.

```
(<operator> <operand1> <operand2> ...)
```

Call expressions in Scheme work exactly like they do in Python. To evaluate them:

1. Evaluate the operator to get a procedure.
2. Evaluate each of the operands from left to right.
3. Apply the value of the operator to the evaluated operands.

For example, consider the call expression `(+ 1 2)`. First, we evaluate the symbol `+` to get the built-in addition procedure. Then we evaluate the two operands `1` and `2` to get their corresponding atomic values. Finally, we apply the addition procedure to the values `1` and `2` to get the return value `3`.

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
```

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

- `+`, `-`, `*`, `/`
- `=`, `>`, `>=`, `<`, `<=`
- `quotient`, `modulo`, `even?`, `odd?`

Questions

3.1 What would Scheme display?

```
scm> (define a (+ 1 2))
      a
scm> a
      3
scm> (define b (+ (* 3 3) (* 4 4)))
      b
scm> (+ a b)
      28
scm> (= (modulo 10 3) (quotient 5 3))
      #t
scm> (even? (+ (- (* 5 4) 3) 2))
      #f
```

4 Special Forms

Special form expressions contain a **special form** as the operator. Special form expressions *do not* follow the same rules of evaluation as call expressions. Each special form has its own rules of evaluation – that's what makes them special!

If Expression

An **`if`** expression looks like this:

```
(if <predicate> <if-true> [if-false])
```

`<predicate>` and `<if-true>` are required expressions and `[if-false]` is optional.

The rules for evaluation are as follows:

1. Evaluate `<predicate>`.
2. If `<predicate>` evaluates to a truth-y value, evaluate `<if-true>` and return its value. Otherwise, evaluate `[if-false]` if provided and return its value.

This is a special form because not all operands will be evaluated! Only one of the second and third operands is evaluated, depending on the value of the first operand.

Remember that only `#f` is a false-y value in Scheme; everything else is truth-y.

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

Boolean Operators

Like Python, Scheme also has the boolean operators **`and`**, **`or`**, and **`not`**. **`and`** and **`or`** are special forms because they are short-circuiting operators.

- `and` takes in any amount of operands and evaluates these operands from left to right until one evaluates to a false-y value. It returns that first false-y value. If there are no false-y values, it returns the value of the last expression (or `#t` if there are no operands)
- `or` also evaluates any number of operands from left to right until one evaluates to a truth-y value. It returns that first truth-y value. If there are no truth-y values, it returns the value of the last expression (or `#f` if there are no operands)
- `not` takes in a single operand, evaluates it, and returns its opposite truthiness value. Note that `not` is a regular procedure and not a special form.

Important note: the only falsey value in scheme is `#f`*. In particular, 0 is* ***truthy****!*

```
scm> (and 25 32)
32
scm> (or 1 (/ 1 0))    ; Short-circuits
1
```

```
scm> (not (odd? 10))
#t
```

Questions

4.1 What would Scheme display?

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

#t 1

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

10

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

-96

```
scm> (if 0 1 2)
```

1

Lambdas and Defining Functions

All Scheme procedures are lambda procedures. One way to create a procedure is to use the `lambda` special form.

```
(lambda (<param1> <param2> ...) <body>)
```

This expression creates a lambda function with the given parameters and body, but does not evaluate the body. Just like in Python, the body is not evaluated until the function is called and applied to some argument values. The fact that neither operand is evaluated is what makes `lambda` a special form.

Another similarity to Python is that lambda expressions do not assign the returned function to any name. We can assign the value of an expression to a name with a `define` special form.

For example, `(define square (lambda (x) (* x x)))` creates a lambda procedure that squares its argument and assigns that procedure to the name `square`.

The second version of the `define` special form is a shorthand for this function definition:

```
(define (<name> <param1> <param2 ...>) <body>)
```

This expression creates a function with the given parameters and body *and* binds it to the given name.

```
scm> (define square (lambda (x) (* x x))) ; Bind the lambda function to the name square
square
scm> (define (square x) (* x x))          ; Equivalent to the line above
square
scm> square
(lambda (x) (* x x))
scm> (square 4)
16
```

Questions

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

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

```
(define (factorial x) (if (<= x 1) 1 (* x (factorial (- x 1)))))
```

4.2 Write a function that returns the n^{th} Fibonacci number.

```
(define (fib n)
                        scm> (define (fib n)
                        scm> (if (= n 0) 0
                        scm> (if (= n 1) 1
                        scm> (+ (fib (- n 1)) (fib (- n 2))))))

                        fib
)
                        scm> (fib 0)
scm> (fib 0)
                        0
0
scm> (fib 1)            scm> (fib 1)
1                       1
scm> (fib 10)           scm> (fib 10)
55
                        55
```

5 Pairs and Lists

All lists in Scheme are linked lists. Scheme lists are composed of two element pairs. We define a list as being either

- the empty list, `nil`
- a pair whose second element is a list

As in Python, linked lists are recursive data structures. The base case is the empty list.

We use the following procedures to construct and select from lists:

- `(cons first rest)` constructs a list with the given first element and rest of the list. For now, if `rest` is not a pair or `nil` it will error.
- `(car lst)` gets the first item of the list
- `(cdr lst)` gets the rest of the list

```
scm> nil
()
scm> (define lst (cons 1 (cons 2 (cons 3 nil))))
lst
scm> lst
(1 2 3)
```

```
scm> (car lst)
1
scm> (cdr lst)
(2 3)
scm> (car (cdr lst))
2
scm> (cdr (cdr (cdr lst)))
()
```

The rule for displaying lists is very similar to that for the Link class from earlier in the class's `__str__` method. It prints out the elements in the linked list as if the list has no nested structure. The only difference is that `Link.__str__` uses angle brackets `<>` and scheme uses parens `()`.

```
scm> (cons 1 (cons 2 (cons 3 nil)))
(1 2 3)
scm> (cons 1 (cons (cons 2 (cons 3 nil)) nil))
(1 (2 3))
```

Two other common ways of creating lists is using the built-in `list` procedure or the `quote` special form:

- The `list` procedure takes in an arbitrary amount of arguments. Because it is a procedure, all operands are evaluated when `list` is called. A list is constructed with the values of these operands and is returned.
- The `quote` special form takes in a single operand. It returns this operand exactly as is, without evaluating it. Note that this special form can be used to return any value, not just a list.
- Similarly, a `quasiquote`, denoted with a backtick symbol, returns an expression without evaluating it. However, parts of that expression can be unquoted, denoted using a comma, and those unquoted parts are evaluated.

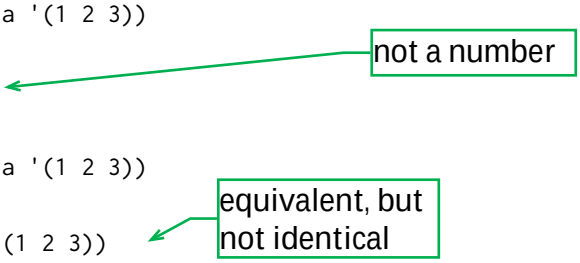
```
scm> (define x 2)
scm> (define y 3)
scm> (list 1 x 3)
(1 2 3)
scm> (quote (1 x 3))
(1 x 3)
scm> '(1 x 3)   ; Equivalent to the previous quote expression
(1 x 3)
scm> '(+ x y)
(+ x y)
scm> `(+ x y)   ; Quasiquotation
(+ x y)
scm> `(+ ,x y)  ; Unquoted with a comma
(+ 2 y)
scm> `(+ ,x ,y)
(+ 2 3)
scm> (eval `(+ ,x ,y))   ; The eval function evaluates the operands
```

5

=, 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 `car`s are `equal?` and their `cdr`s are `equal?`(that is, they have the same contents). Otherwise, `equal?` behaves like `eq?`.

```
scm> (define a '(1 2 3))
a
scm> (= a a)
Error
scm> (equal? a '(1 2 3))
#t
scm> (eq? a '(1 2 3))
#f
scm> (define b a)
b
scm> (eq? a b)
#t
```



Questions

5.1 Write a function which takes two lists and concatenates them.

Notice that simply calling `(cons a b)` would not work because it will create a deep list. Do not call the builtin procedure `append`, which does the same thing as `my-append`.

```
(define (my-append a b)
```

```
scm> (define (my-append a b)
scm> (if (null? a) b
scm> (cons (car a) (my-append (cdr a) b))))

my-append

scm> (my-append '(1 2 3) '(2 3 4))

(1 2 3 2 3 4)
```

```
)
scm> (my-append '(1 2 3) '(2 3 4))
(1 2 3 2 3 4)
```

5.2 Write a Scheme function that, when given an element, a list, and an index, inserts the element into the list at that index.

```
(define (insert element lst index)
```

```
scm> (define (insert element lst index)
scm>   (if (= index 0)
scm>       (cons element lst)
scm>       (cons (car lst) (insert element (cdr lst) (- index 1)))))
scm> (define lst '(1 2 3 2 3 4))
scm> (insert 5 lst 3)

insert,lst,(1 2 3 5 2 3 4)   remember to quote the lst!

scm> (define (insert element lst index)
scm>   (if (= index 0)
scm>       (cons element lst)
scm>       (cons (car lst) (insert element (cdr lst) (- index 1)))))
scm> (define lst (1 2 3 2 3 4))
scm> (insert 5 lst 3)

insert,Error: cannot call: 1,(1 2 3 5 2 3 4)
```

5.3 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)
```

```
scm> (define (duplicate lst)
scm> (if (null? lst) '()
scm> (cons (car lst) (cons (car lst) (duplicate (cdr lst))))))
scm> (define lst '(1 2 3 4))
scm> (duplicate lst)

duplicate,lst,(1 1 2 2 3 3 4 4)
```

```
)
```