CS 61A
Summer 2018

# Scheme

Discussion 7: June 12, 2018 Solutions

## 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 Scheme Expressions

Primitive expressions in Scheme include self-evaluating expressions, like numbers and booleans, and symbols.

All non-primitive expressions in Scheme are **combinations**. These are expressions that combine other expressions in a set of parentheses like so:

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

where `<operator>`, `<operand1>`, `<operand2>` are all Scheme expressions. A combination can have zero or more operands, but it must have an operator.

Combinations are one of two things:

1. **Call expressions**, if the `<operator>` is a procedure.
2. **Special form expressions**, if the `<operator>` is a special form.

### 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

2.1 What would Scheme display?

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

```
1
```

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

```
3
```

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

```
25
```

```
scm> (+ 1 (modulo 10 3) (quotient 10 3))
```

```
5
```

```
scm> (> (+ 2 3) (* 2 3))
```

```
#f
```

```
scm> (even? (quotient (* 4 6) 2))
```

```
#t
```

```
scm> (= (+ (- (* 5 4) 3) 2) 1)
```

```
#f
```

## Special Form Expressions

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!

The special forms we'll see in today's discussion include `if` `and`, `or`, `lambda`, and `define`.

## If Expression

The `if` special form allows for control flow in our programs. An **`if`** expression looks like this:

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

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

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
scm> (if (= (+ 1 2) 3) 'equal)
equal
```

## 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 either returns that first false-y value or the value of the last expression if all operands evaluate to a truth-y value.
- `or` takes in any amount of operands and evaluates these operands from left to right until one evaluates to a truth-y value. It either returns that first truth-y value or the value of the last expression if all operands evaluate to a false-y value.

- `not` takes in a single operand, evaluates it, and returns its opposite truthiness value. Because its one operand is always evaluated, `not` is a regular procedure rather than a special form.

```
scm> (and 25 32)
32
scm> (or #f (< 4 3) (= 1 2))
#f
scm> (or 1 2)    ; Short-circuit
1
scm> (and #f 2)  ; Short-circuit
#f
scm> (not (odd? 10))  ; Not a special form!
#t
```

## Questions

2.1 What would Scheme display?

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

```
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
```

## Lambda and Define

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 expression is not evaluated until the lambda function is called and thus 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 the `define` special form:

```
(define <name> <expr>)
```

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

2.2 Write two Scheme expressions that are equivalent to the following Python statement:

```
cat = lambda meow, purr: meow + purr
```

```
    (define cat (lambda (meow purr) (+ meow purr)))
    (define (cat meow purr) (+ meow purr))
```

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

```
(define (fib n)


    (if (or (= n 0) (= n 1))
        n
(+ (fib (- n 1)) (fib (- n 2)))))
```

# 3 Pairs and Lists

## Pairs

Scheme pairs are compound data types that hold exactly two elements. Pairs are displayed in the interpreter with a dot in between the two elements.

Here are the procedures to construct and select from pairs:

- `(cons x y)` constructs a pair with elements `x` and `y`
- `(car p)` gets the first element of a pair `p`
- `(cdr p)` gets the second element of a pair `p`

```
scm> (define a (cons 2 3))
a
scm> a
(2 . 3)
scm> (car a)
2
scm> (cdr a)
3
```

## Lists

Scheme lists implement an abstract data type known as a linked list with a sequence of pairs. We define a list to be one of two things:

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

Linked lists are recursive data structures. The base case is the empty list.

We can use the same procedures used to construct and select from pairs to construct and select from lists:

- `(cons first rest)` constructs a list with the given first element and rest of the list
- `(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)))
()
```

In general, the rule for displaying a pair is as follows: a dot separates the `car` and `cdr` fields of a pair. However, 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)`. Intuitively, a dot will be followed by an open parenthesis if the `cdr` of a pair is another pair.

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

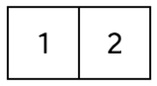
```
scm> (list 1 2 3)
(1 2 3)
scm> (quote (1 2 3))
(1 2 3)
scm> '(1 2 3)   ; Equivalent to the previous quote expression
(1 2 3)
```

To check if a list is a well-formed list (i.e. it fits one of the descriptions in the definition above), use the predicate procedure `list?`.

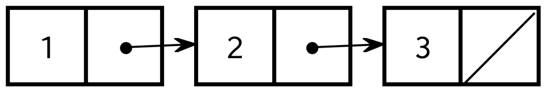
## Box-and-pointer Diagrams

We can draw box-and-pointer diagrams to help us visualize Scheme pairs and lists.

A pair is represented as two boxes, each containing one element of the pair. For example, to draw the pair constructed by `(cons 1 2)`:



To draw well-formed lists, draw a series of pairs connected by arrows. Specifically, the first item in each pair corresponds to an item in the list. The second item in each pair is an arrow to the rest of the list. Here is the list `(1 2 3)`:



## Questions

3.1 What would Scheme display?

```
scm> (cons 10 (cons 11))
```

```
Error
```

```
scm> (car (cons 10 (cons 11 nil)))
```

```
10
```

```
scm> (cdr (cons 10 (cons 11 nil)))
```

```
(11)
```

```
scm> (cons 5 '(6 7 8))
```

```
(5 6 7 8)
```

```
scm> '(1 . (2 3))
```

```
(1 2 3)
```

```
scm> (define a 10)
a
scm> (list 8 9 a 11)    ; list procedure evaluates all operands
```

```
(8 9 10 11)
```

```
scm> '(8 9 a 11)        ; quote special form does not evaluate operand
```

```
(8 9 a 11)
```

```
scm> (list? (cons 1 2))
```

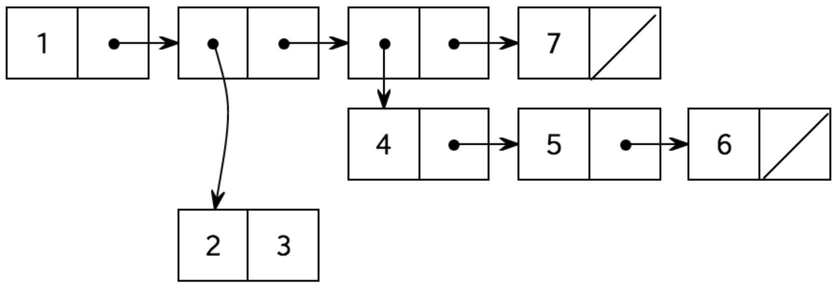
```
#f
```

```
scm> (list? (cons 1 (cons 2 '())))
```

```
#t
```

3.2 Draw out a box-and-pointed diagram for the following list:

```
scm> (define nested-lst (list 1 (cons 2 3) '(4 5 6) 7))
nested-lst
```



Then, write out what Scheme would display for the following expressions:

```
scm> (cdr nested-lst)
```

```
((2 . 3) (4 5 6) 7)
```

```
scm> (cdr (car (cdr nested-lst)))
```

```
3
```

```
scm> (cons (car nested-list) (car (cdr (cdr nested-list))))
```

```
(1 4 5 6)
```

3.3 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.

```
(define (concat a b)


  (if (null? a)
    b
    (cons (car a) (concat (cdr a) b))))
```

Video walkthrough

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

3.4 Write a function that takes an element `x` and a non-negative integer `n`, and returns a list with `x` repeated `n` times.

```
(define (replicate x n)


    (if (= n 0)
        nil
        (cons x (replicate x (- n 1)))))
```

Video walkthrough

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

3.5 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 function that takes a compressed sequence and expands it into the original sequence. *Hint:* You may want to use `concat` and `replicate`.

```
(define (uncompress s)


  (if (null? s)
    s
    (concat (replicate (car (car s)) (car (cdr (car s))))
            (uncompress (cdr s)))))
```

Video walkthrough

```
scm> (uncompress '((a 1) (b 2) (c 3)))
(a b b c c c)
```

3.6 Write a function that takes a procedure and applies it to every element in a given list.

```
(define (map fn lst)


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

scm> (map (lambda (x) (* x x)) '(1 2 3))
(1 4 9)
```

3.7 Write a function that takes a procedure and applies 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 `list?` procedure to detect whether a value is a list.

```
(define (deep-map fn lst)


  (cond ((null? lst) lst)
        ((list? (car lst)) (cons (deep-map fn (car lst)) (deep-map fn (cdr lst))))
        (else (cons (fn (car lst)) (deep-map fn (cdr 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)
```

## 4 Extra Questions

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

```
(define (make-tree label branches) (cons label branches))

(define (label tree)

(define (branches tree)


(define (label tree) (car tree))
(define (branches tree) (cdr tree))
```

4.2 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, and also write a helper function for summing up the entries of a list.

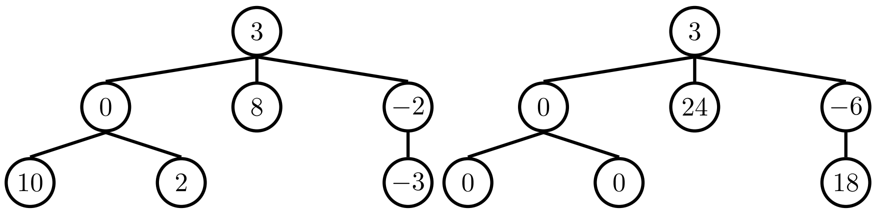
```
(define (tree-sum tree)

    (+ (label tree) (sum (map tree-sum (branches tree)))))


(define (sum lst)
    (if (null? lst) 0 (+ (car lst) (sum (cdr lst)))))
```

4.3 Using the abstract data type above, write a 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 a helper function that keeps track of the current product.



```
(define (path-product-tree t)

    (define (path-product t product)
        (let ((prod (* product (label t))))
            (make-tree prod
                (map (lambda (t) (path-product t prod))
                     (branches tree)))))
    (path-product t 1))
```