CS 61A
Summer 2017

Scheme
Discussion 10: July 25, 2017

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 Define

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 What would Scheme print?

```
scm> (define a 1)


scm> a


scm> (define b a)


scm> b


scm> (define c 'a)


scm> c
```

3 Call Expressions

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.

```
scm> (- 1 1)                ; 1 - 1
0
scm> (/ 8 4 2)              ; 8 / 4 / 2
1
scm> (* (+ 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 What would Scheme print?

```
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.

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 25 32)
32
scm> (or 1 2)
1
```


```
scm> (and 25 32)
32
scm> (or 1 2)
1
```


4.1 What does Scheme print?

```
scm> (and 0 2 200)


scm> (or True (/ 1 0))


scm> (and False (/ 1 0))


scm> (not 3)
```

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
```

4.1 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)
```

Lambda Expressions

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.

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
```

Let Form

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

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

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

```
scm> (define x 3)
x
scm> (define y 4)
y
scm> ((lambda (x y) (+ x y)) 6 7)
13
```

This special form is really just equivalent to:

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

Questions

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

```
(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 `cons` (which takes two arguments). `nil` represents the empty list. If you have a linked list in Scheme, you can use selector `car` to get the first element and selector `cdr` to get the rest of the list. (`car` and `cdr` don't stand for anything anymore, but if you want the history go to 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 `cons` always takes exactly 2 arguments. These lists are called **malformed list**. The difference is a dot:

```
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)
```

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, `(0 . (1 . 2))` becomes `(0 1 . 2)`

There are many useful operations and shorthands on lists. One of them is `list` special form `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.

```
scm> (list 1 2 3)
(1 2 3)
scm> '(1 2 3)
(1 2 3)
scm> (car '(1 2 3))
1
scm> (equal? '(1 2 3) (list 1 2 3))
#t
scm> '(1 . (2 3))
(1 2 3)
scm> '(define (square x) (* x x))
(define (square x) (* x x))
```

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

5.2 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> (replicate 5 3)
(5 5 5)
```

6 Tail-Call Optimization

Scheme implements tail-call optimization, which allows programmers to write recursive functions that use a constant amount of space. A **tail call** occurs when a function calls another function as its **last action of the current frame**. Because in this case Scheme won't make any further variable lookups in the frame, the frame is no longer needed, and we can remove it from memory. In other words, if this is the last thing you are going to do in a function call, we can reuse the current frame instead of making a new frame.

Consider this version of `factorial` that does *not* use tail calls:

```
(define (fact n)
  (if (= n 0) 1
      (* n (fact (- n 1)))))
```

The recursive call occurs in the last line, but it is not the last expression evaluated. After calling `(fact (- n 1))`, the function still needs to multiply that result with `n`. The final expression that is evaluated is a call to the multiplication function, not `fact` itself. Therefore, the recursive call is *not* a tail call.

However, we can rewrite this function using a helper function that remembers the temporary product that we have calculated so far in each recursive step.

```
(define (fact n)
  (define (fact-tail n result)
    (if (= n 0) result
        (fact-tail (- n 1) (* n result))))
  (fact-tail n 1))
```

`fact-tail` makes a single recursive call to `fact-tail` that is the last expression to be evaluated, so it is a tail call. Therefore, `fact-tail` is a tail recursive process. Tail recursive processes can take a constant amount of memory because each recursive call frame does not need to be saved. Our original implementation of `fact` required the program to keep each frame open because the last expression multiplies the recursive result with `n`. Therefore, at each frame, we need to remember the current value of `n`.

In contrast, the tail recursive `fact-tail` does not require the interpreter to remember the values for `n` or `result` in each frame. Instead, we can just *update* the value of `n` and `result` of the current frame! Therefore, we can carry out the calculation using only enough memory for a single frame.

6.1 Identifying tail calls

A function call is a tail call if it is in a **tail context** (but a tail call might not be a recursive tail call as seen above in the first `fact` definition). Tail context simply means the expression is the last to be evaluated in that form. For example, we consider the following to be tail contexts:

- the last sub-expression in a lambda's body
- the second or third sub-expression in an `if` form

- any of the non-predicate sub-expressions in a `cond` form
- the last sub-expression in an `and` or an `or` form
- the last sub-expression in a `begin`'s body

These make sense intuitively because the last expression to be evaluated in an `if` form is not the condition, but rather either the second or third sub-expressions which are evaluated depending on if the condition is True or False. See if you can reason through why the others above are considered in tail context as well.

Before we jump into questions, a quick tip for defining tail recursive functions is to use helper functions. A helper function should have all the arguments from the parent function, plus additional arguments like `total` or `counter` or `result`.

Questions

6.1 For each of the following functions, identify whether it contains a recursive call in a tail context. Also indicate if it uses a constant number of frames.

```
(define (question-a x)
  (if (= x 0)
      0
      (+ x (question-a (- x 1)))))


(define (question-b x y)
  (if (= x 0)
      y
      (question-b (- x 1) (+ y x))))


(define (question-c x y)
  (if (> x y)
      (question-c (- y 1) x)
      (question-c (+ x 10) y)))


(define (question-d n)
  (if (question-d n)
      (question-d (- n 1))
      (question-d (+ n 10))))
```

6.2 Write a tail recursive function, `sum`, that takes in a Scheme list and returns the numerical sum of all values in the list.

```
(define (sum lst)
```

7 Extra Practice

7.1 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.

```
(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
```

7.2 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)
```

```
)
```