SCHEME AND CALCULATOR 5b

COMPUTER SCIENCE 61A

July 25, 2013

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.

This discussion will give you some more practice with Scheme and introduce you to interpreting computer programs.

0.1 Warmup — What Would Scheme Do?

```
STk> (define a 1)
?
STk> a
?
STk> (define b a)
?
STk> b
?
STk> (define c 'a)
?
STk> c
?
STk> (define (foo x) (+ x 1))
?
STk> (define (bar x) (* x (+ x 1)))
?
STk> (bar (foo 3))
?
```

1 Evaluating Function Calls and Special Forms

1.1 Functions

Now, you might notice that Scheme does function calls differently. To call a function in Scheme, first give the symbol for the function name, then give the arguments (remember the spaces!). But just as in Python, you evaluate the operator (the leftmost expression between the parentheses) before evaluating the arguments left to right. Then, apply the operator to the evaluated arguments.

So when you evaluate `(+ 1 2)`, first evaluate the `+` symbol which is bound to a built-in addition function. Then, evaluate the primitives `1` and `2`. Finally, apply the addition function to the arguments.

Some important functions you'll want to use are:

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

1.2 Questions

1. What do the following return?
   - `(+ 1)`
   - `(* 3)`
   - `(= (+ 2 1) (* 1.5 2))`

1.3 Special Forms

However, there are certain things that look like function calls that aren't. These are called special forms and have their own rules for evaluation. You've already seen one- `define` where, of course, the first argument can't be evaluated (or else it'd search for unbound variables!). Another one we'll use for this class is `if`.

An `if` expression looks like this: `(if <CONDITION> <THEN> <ELSE>)`, where `<CONDITION>`, `<THEN>`, and `<ELSE>` are expressions.

It's evaluated exactly as it is in Python. First, the `<CONDITION>` is evaluated. If it evaluates to `#f`, then `<ELSE>` is evaluated. Otherwise, `<THEN>` is evaluated. Everything that is not `#f` is a "true" expression.

```
STk> (if 'this-evaluates-to-true 1 2)
1
STk> (if #f (/ 1 0) 'this-is-returned)
this-is-returned
```

There are also special forms for the boolean operators which exhibit the same short-circuiting behavior that you see in Python. The return values are either the value that lets you know the expression evaluates to a true value or `#f`.

```
STk> (and 1 2 3)
3
STk> (or 1 2 3)
1
STk> (or #t (/ 1 0))
#t
STk> (and #f (/1 0))
#f
STk> (not 3)
#f
STk> (not #t)
#f
```

1.4 Questions

```
STk> (if (or #t (/ 1 0)) 1 (/ 1 0))
?
STk> (if (> 4 3) (+ 1 2 3 4) (+ 3 4 (* 3 2)))
?
STk> ((if (< 4 3) + -) 4 100)
?
```

2 Lambdas, Environments, and Defining Functions

Scheme has lambdas too! In fact, lambdas are more powerful in Scheme than in Python. The syntax is `(lambda (<PARAMETERS>) <EXPR>)`. Like in Python, lambdas are function values. Likewise, in Scheme, when a lambda expression is called, a new frame is created where the symbols defined in the `<PARAMETERS>` section are bound to the arguments passed in. Then, `<EXPR>` is evaluated under this new frame. Note that `<EXPR>` is not evaluated until the lambda value is called.

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

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

```
STk> (define square (lambda (x) (* x x)))
square
STk> (square 4)
16
```

You might notice that this is a little tedious though. Luckily Scheme has a way out- `define`:

```
STk> (define (square x) (* x x))
square
STk> (square 5)
25
```

When you do `(define (<FUNCTIONNAME> <PARAMETERS>) <EXPR>)`, Scheme will automatically transform it to `(define <FUNCTIONNAME> (lambda (<PARAMETERS>) <EXPR>)` for you. In this way, lambdas are more foundational to Scheme than they are to Python. Unlike Python lambdas, Scheme lambdas can have more than one statement.

There is also another 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>)
```

You'll notice that what `let` does then is bind symbols to expressions. For example, this is useful if you need to reuse a value multiple times, or if you want to make your code more readable:

```
(define (sin x)
    (if (< x 0.000001)
        x
```

```
        (let ( (recursive-step (sin (/ x 3))) )
            (- (* 3 recursive-step)
               (* 4 (expt recursive-step 3))))))
```

2.1 Questions

1. Write a function that calculates factorial. (Note how you haven't been told any methods for iteration.)

```
(define (factorial x)



                                                    )
```

2. Write a function that calculates the Nth fibonacci number

```
(define (fib n)
    (if (< n 2)
        1

                                                    )
```

3 Pairs and Lists

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 from which you'll build most complex data structures out of is the `pair`. A pair is an abstract data type that has the constructor `cons` which takes two arguments, and it has two accessors `car` and `cdr` which get the first and second argument respectively. `car` and `cdr` don't stand for anything really now but if you want the history go to http://en.wikipedia.org/wiki/CAR_and_CDR

```
STk> (define a (cons 1 2))
a
STk> a
(1 . 2)
STk> (car a)
1
STk> (cdr a)
2
```

Note that when a `pair` is printed, the `car` and `cdr` element are separated by a period.

A common data structure that you build out of pairs is the list. A list is either the empty list whose literal is `'()`, also known as `nil`, another primitive, or it's a cons pair where the `cdr` is a list. (Note the similarity to `Rlists`!)

```
STk> '()
()
STk> nil
()
STk> (cons 1 (cons 2 nil))
(1 2)
STk> (cons 1 (cons 2 (cons 3 nil)))
(1 2 3)
```

Note that there are no dots here. That is because when a dot is followed by the left parenthesis, the dot and the left parenthesis are deleted; when a left parenthesis is deleted, its matching right parenthesis is deleted also. You can check if a list is nil by using the `null?` function.

A shorthand for writing out a list is:

```
STk> '(1 2 3)
(1 2 3)
STk> '(define (square x) (* x x))
(define (square x) (* x x))
```

You might notice that the return value of the second expression looks a lot like Scheme code. That's because Scheme code is made up of lists. When you use the single quote, you're telling Scheme not to evaluate the list, but instead keep it as just a list.

This is one of the reasons why Scheme is so cool — it can be defined within itself!

3.1 Questions

1. Define `map` where the first argument is a function and the second a list. This should work like Python's `map`.

   ```
   (define (map fn lst)
   ```

)

2. Define `reduce` where the first argument is a function that takes two arguments, the second a default value and the third a list. This should work like Python's `reduce`.

```
(define (reduce fn s lst)



                                                    )
```

4 Calculator

In lecture, you saw how to implement the Calculator language using regular Python. The Calculator is a Scheme-like language that can handle the four basic arithmetic operations. These operations can be nested and can take varying numbers of arguments. Our goal now is to prepare for Project 4 by understanding the pieces of the Calculator interpreter.

4.1 Representing Expressions

There are two kinds of expressions. A **call expression** is a Scheme list where the first element is the operator and each of the remaining elements is an operand. A **primitive expression** is an operator symbol or number. When we type a line at the Calculator prompt and hit enter, we've just sent an expression to the interpreter.

To represent Scheme lists in Python, we'll be using `Pair` objects. `Pair`s are just like the Rlists you've come to know and love!

4.2 Questions

1. Translate the following Python representation of Calculator expressions into the proper Scheme representation:

```
>>> Pair('+', Pair(1, Pair(2, Pair(3, Pair(4, nil)))))
(+ 1 2 3 4)
>>> Pair('+', Pair('1', Pair(Pair('*', Pair(2, Pair(3, nil))),
                             nil)))

>>> Pair('*', Pair(Pair('*', Pair(1, Pair(2, nil))), Pair(3, nil)))
```

2. Translate the following Calculator expression into calls to the `Pair` constructor:

```
> (+ 1 2 (- 3 4))
```

4.3 Evaluating and Applying

Evaluation discovers the form of an expression and executes the corresponding evaluation rule.

Primitive expressions are evaluated directly. Call expressions are evaluated recursively as we've seen before. First, evaluate the operator, then the operands left to right. Then, collect their values as a list of arguments and apply the operator to those arguments.

If you refer to the `minicalc.py` file, you can see that all we've done in the `calc_eval` function is follow the rules of evaluation that were just outlined! If the expression is primitive (not a `Pair`), simply return it. Otherwise, evaluate the operands and apply the operator to the evaluated operands.

We **apply** the operator with the `calc_apply` function, which is a dispatch function that will dispatch on the operator name. Depending on what the operator is, we can match it to a corresponding Python call. Each conditional clause above handles the application of one operator.

One way to remember the two functions: `calc_eval` deals with **expressions**, whereas `calc_apply` deals with **values**.

4.4 Questions

1. Suppose we typed each of the following expressions into the Calculator interpreter. How many calls to `calc_eval` would they each generate? How about `calc_apply`?

```
> (+ 2 4 6 8)

> (+ 2 (* 4 (- 6 8)))
```

2. We also want to be able to perform division, as in `(/ 4 2)`. Supplement the existing code to handle this. If division by 0 is attempted, you should raise a ZeroDivisionError. (Hint: a helper function that does something like Python's `in` operator for a Scheme list may be helpful here!)

```
def scheme_in(elem, scheme_lst):
```

```
def calc_apply(op, args):
        ...
        if op == '/':
```

3. Alyssa P. Hacker and Ben Bitdiddle are also tasked with implementing the `and` operator, as in `(and (= 1 2) (< 3 4))`. Ben says this is easy: they just have to follow the same process as in implementing `*` and `/`. Alyssa is not so sure. Who's right?

4. Now that you've had a chance to think about it, you decide to try implementing `and` yourself. You may assume the conditional operators (e.g. <, >, =, etc) have already been implemented for you.