SCHEME AND EXCEPTIONS 11

COMPUTER SCIENCE 61A

April 4th, 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 practice with the Scheme language, in addition to going over the idea of exception handling in Python.

1 Scheme Warmup

Let's do some practice Scheme questions!

1.1 Questions

1. What gets printed?

```
STk> (define a 1)
?
STk> a
?
STk> (define b a)
?
STk> b
?
STk> (define c 'a)
?
STk> c
?
```

2 Evaluating Function Calls and Special Forms

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

```
STk> (+ 1 2)
3
STk> (- 2 3)
-1
STk> (* 6 3)
18
STk> (/ 5 2)
2.5
STk> (+ 1 (* 3 4))
13
```

2.1 Functions

Now you might notice that **Scheme** does function calls differently. In **Scheme**, to a call a function, you give the symbol for the function name first, then you give the arguments (remember the spaces). But just as in Python, what you do is you evaluate the operator (the first expression to the right of the `(`), then you evaluate each of the arguments and then apply those evaluated arguments to the thing returned by the first expression. So when you evaluate `(+ 1 2)`, you evaluate the `+` symbol which is bound to a built-in addition function, then you evaluate `1` and `2`. Finally, you apply `1` and `2` to the function value bound to `+`.

Some important functions you'll want to use are:

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

2.2 Questions

1. What do the following return?
   - `(+ 1)`
   - `(* 3)`
   - `(+ (* 3 3) (* 4 4))`
   - `(define a (define b 3))`

2.3 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. How it gets evaluated however is that first, `<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
```

2.4 Questions

1. What does the following do?

```
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)
?
```

3 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**.

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))))))
```

3.1 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

                                                          )
   ```

4 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 why **Scheme** is cool. It can be defined within itself.

4.1 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)




                          )
```

3. Fill out the following to complete an abstract type for binary trees:

```
(define (make-btree entry left right)
    (cons entry (cons left right)))

(define (entry tree)
                          )

(define (left tree)
                          )

(define (right tree)
                          )
```

4. Using the above definition, write a function that sums over the binary tree. For our purposes, assume that if there is no left or if there is no right branch, the values for those should return 0.

```
(define (btree-sum tree)

)
```

5 Exceptions

Up to this point in the semester, we have assumed that the input to our functions are always correct, and thus have not done any error handling. However, functions can often have large domains, and we want our functions to handle erroneous input gracefully. This is where exceptions come in.

Exceptions provide a general mechanism for adding error-handling logic to programs. **Raising an exception** is a technique for interrupting the normal flow of execution in a program, signaling that some exceptional circumstance has arisen.

An exception is an instance of a class that inherits, either directly or indirectly, from the `BaseException` class. The following is an example of how to raise an exception:

```
>>> raise Exception('an error occurred!')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
Exception: An error occurred!
```

Notice how the string "An error occurred" is an argument to the `Exception` object being created, and the string is part of what Python prints out in response to the exception being raised.

If the exception is raised while within a `try` statement, then the interpreter will immediately look for an `except` clause that handles the type of exception being raised. `try` and `except` clause allow programs to respond to unexpected arguments and other errors gracefully, rather than terminating entirely.

Heres how to structure `try` and `except` clauses:

```
try:
    <try suite>
except <exception class> as <name>:
    <except suite>
except <exception class> as <name>:
    <except suite>
```

5.1 Questions

1. Fill in all the blanks to produce the desired output:

```
>>> try:
        x = _______
    except ___________________ as e:
        print('handling a', type(e))
        x = _______
handling a <class ZeroDivisionError>
>>> x
9001
```

2. Write the function `safe_square` that uses exceptions to print "Incorrect argument type' and return None when anything other than an `int` or `float` class is given as an argument. Otherwise, `safe_square` should multiply the argument by itself. A useful fact is that a TypeError is raised when * is given incorrect arguments.

```
def safe_square(x):
```

3. Predict the output of each of the following lines, assuming `safe_square` is implemented as described in the previous question.

```
>>> safe_square('hello')

>>> safe_square('hello' * 5)

>>> safe_square('hello' * 'hello')

>>> safe_square(1 * 2.5)

>>> safe_square(1 / 0)
```