CS61A Lab 10: Interpreters

Week 10, 2011

We're beginning our journey into the world of interpreters! You may have seen an interpreter before. (Hint: that thing you've been running all your Python code in? Yeah, that's an interpreter). In fact, the Python interpreter we've been using all semester long is really nothing but a program, albeit a very complex one. You can think of it as a program that takes strings as input and produces their evaluated result as output.

There's nothing magical about interpreters, though! They're programs, just like the ones you've been writing throughout the entire semester. In fact, it's possible to write a Python interpreter in Python; PyPy is proof of that. However, because Python is a complex language, writing an interpreter for it would be a very daunting project. Instead, we'll play around with an interpreter for a calculator language. We'll also start to get familiar with the language Scheme, which you will write an interpreter for in project 4. (This course used to be taught in Scheme!)

Calc: our first interpreter!

In lecture, you were introduced to our first interpreter program, `calc`, which acts as a simple calculator. It currently supports basic arithmetic, if statements, and lambda statements. You can copy the code to your current directory by running the following in your terminal:

```
cp ~cs61a/lib/lab/lab10/calc.py .
```

You can also find the code here. You can try running `calc` by running this command in terminal:

```
python3 calc.py
```

To exit the program, type Ctrl-d.

The following questions in the lab will ask you to look through the code to understand how it works and to modify the code to add your own extensions to the current version of the `calc` language.

Question 1:

Trace through the code in `calc.py` that would be evaluated when you type the following into `calc`.

```
calc> 2
calc> add(2, 3)
calc> add(2, 3, 4)
calc> +(2, 3)
calc> add(2)
calc> add(2, mul(4, 5))
```

Explain to another student how each statement is evaluated.

Question 2:

It's boring to only do simple math, it'd be nice to compare numbers and eventually make choices based on these comparisons. Add in the 3 additional operators `eq`, `lt`, and `gt` which allow us to compare two numbers. It should work in the following way:

```
calc> eq(5, 6)
False
calc> eq(4, 4)
True
calc> lt(5, 6)
True
calc> lt(6, 5)
False
calc> lt(6, 6)
False
calc> gt(5, 6)
```

```
False
calc> gt(6, 5)
True
calc> gt(6, 6)
False
calc> eq(mul(2, 2), 4)
True
calc> eq(mul(2, 6), 18)
False
calc> gt(mul(2, 2), 5)
False
calc> lt(mul(2, 2), 5)
True
```

(*Hint:* You'll probably want to edit both the `known_operators` list and the `calc_apply` function.)

Question 3:

Now that we can compare values, it'd be great to be able to make decisions using something like Python's `if` statement. Let's add `if` to the `calc` language!

To do this, we need to change the way `calc` **evaluates** the "arguments" of `if` to be different from the way it handles arguments for other functions. As a result, you will be making changes to `calc_eval` in order to get `if` working correctly. Once you've implemented `if`, you should be able to do the following:

```
calc> if(gt(5, 6), mul(5, 5), mul(6, 6))
36
calc> if(eq(6, 6), mul(5, 5), mul(6, 6))
25
calc> if(eq(6, 6), mul(5, 5), div(6, 0))
25
calc> if(eq(5, 6), mul(5, 5), div(6, 0))
ZeroDivisionError: division by zero
```

Note: you should make sure that there's no error for that third example! `if` doesn't evaluate all three of its arguments!

Question 4:

Now we will add in `and` and `or` which will behave similarly to `and` and `or` in Python. Remember that `and` and `or` are short-circuiting operators and that they return the *value* of the last expression they evaluate before stopping! Once you're done it should behave like the examples below:

```
calc> and(eq(2, 3), div(4, 0))
False
calc> and(eq(2, 2), div(4, 0))
ZeroDivisionError: division by zero
calc> and(4, 5)
5
calc> or(eq(2, 3), div(4, 0))
ZeroDivisionError: division by zero
calc> or(eq(2, 2), div(4, 0))
True
calc> or(4, 5)
4
calc> or(eq(2, 3), eq(2, 2), div(4, 0))
True
calc> and(eq(2, 2), eq(2, 3), div(4, 0))
False
```

Welcome to Scheme!

Scheme is a programming language. It is a dialect of Lisp, an even *older* language, that is famous for having lots of parentheses. Much like Python, we also have an interpreter for Scheme installed on the lab machines, which you can access by running `stk` on the lab machines. Doing so gives you a prompt, which we can play around with. It is important to be familiar with Scheme in order for you to complete project 4, where you write an interpreter for Scheme in Python. So let's dive in!

We have numbers...

```
STk> 0
0
```

```
STk> 2
2
```

And we have strings...

```
STk> "hello"
"hello"
```

And we have booleans... (`#t` means true, `#f` means false)

```
STk> #f
#f
STk> (= 1 1)
#t
STk> (if (= 1 1) "math is fine" "math is broken")
"math is fine"
```

And we have functions...

```
STk> 1+1
*** Error:
    unbound variable: 1+1
Current eval stack:
__________________
  0     1+1
```

Wait, what gives? Plus doesn't seem to work! That's because Scheme doesn't have infix notation (where the operator appears between the operands). No, in Scheme, every time you call a function, you put the function in *front*, and you enclose the whole thing in parentheses:

```
STk> (+ 1 2)
3
```

This is true of *any* function call you could ever want to make in Scheme. Just type an open parenthesis, the name of your function, any arguments, then a close parenthesis. Behold:

```
STk> (+ 1 (* 3 4))
13
STk> (list 1 2 3)
(1 2 3)
STk> (map (lambda (x) (* x 2)) (list 1 2 3))
(2 4 6)
```

Hey, that last example included our favorite keyword again, `lambda`! It's also present in Scheme, and it does something very similar. In this case, `(lambda (x) (* x 2))` creates a Scheme function that takes a single argument, `x`, and returns `x` multiplied by 2. Note that it looks similar to its Python equivalent, `lambda x: x * 2`, except with more parentheses and less colons!

You can even define your own functions!

```
STk> (define (fib x)
             (if (<= x 1)
                 1
                 (+ (fib (- x 1)) (fib (- x 2)))))
STk> (fib 5)
8
STk> (fib 20)
10946
```

To exit STk, use either the function `bye` or `exit`:

```
STk> (bye)
```

Question 1:

Try each of the following lines of Scheme in STk and see if you can explain to your peers what is happening (do them in order, some of the lines depend on the code above it):

```
3
```

```
(+ 2 3)
```

```
(+ 5 6 7 8)

(+)

(sqrt 16)

(+ (* 3 4) 5)

+

'+

'hello

'(+ 2 3)

'(good morning)

(first 274)

(butfirst 274)

(first 'hello)

(first hello)

(first (bf 'hello))

(+ (first 23) (last 45))

(define pi 3.14159)

pi

'pi

(+ pi 7)
(* pi pi)

(define (square x) (* x x))
```

```
(square 5)

(square (+ 2 3))

(define a 3)

(define b (+ a 1))

(+ a b (* a b))

(= a b)

(if (and (> b a) (< b (* a b)))
    b
    a)

(cond ((= a 4) 6)
      ((= b 4) (+ 6 7 a))
      (else 25))

(+ 2 (if (> b a) b a))

(* (cond ((> a b) a)
         ((< a b) b)
         (else -1))
   (+ a 1))

((if (< a b) + -) a b)
```