INTERPRETATION AND SCHEME LISTS 12

COMPUTER SCIENCE 61A

July 26, 2012

1 Calculator Language

We are beginning to dive into the realm of interpreting computer programs. To do so, we will examine a few new programming languages. The Calculator language, a very simple language and invented for this class, is the first of these examples.

Our Calculator language (or "Calc") looks very similar to Python, except it only handles four arithmetic operations: `add`, `sub`, `mul`, `div`, and the corresponding symbols (`+`, `-`, `*`, and `/`). All of these operators are *prefix* operators, which means that the operator precedes the operands that it is operating on.

Here are a few examples of Calculator in action:

```
calc> 6
6
calc> add(3, 5)
8
calc> mul()
1
calc> add(1, mul(3, sub(3, 7)))
-11
```

Our goal now is to write an interpreter for the Calculator language. The job of an interpreter is, given an expression, perform all the steps necessary to evaluate it. The interpreter that we saw in lecture spends most of its time in the read-eval-print-loop (often called a REPL), which continuously reads a line of input, translates this into an expression object (we call this *parsing the input*), evaluates that expression object, and prints the result. Nearly all interpreters can be organized into this REPL pattern!

We are primarily interested in the E part of REPL, which stands for evaluation.

1.1 Representing and Evaluating Expressions

When we type a line at the Calculator prompt and hit enter, we have just sent an expression to the interpreter. We can represent an expression as an object:

```
class Exp(Object):
    def __init__(self, operator, operands):
        self.operator = operator
        self.operands = operands
    def __repr__(self):
        return "Exp({0}, {1})".format(repr(self.operator),
                                      repr(self.operands)))
    def __str__(self):
        operand_strs = ", ".join(map(str, self.operands))
        return "{0}({1})".format(self.operand, operand_strs)
```

The most important thing to note is that every `Exp` is represented by its operator and operands. Also, another thing to note: the operands can also be `Exp` objects themselves. In fact, this representation of an expression is called an *expression tree*, since the "children" of the operator are also `Exp` objects.

For example, if we wanted to represent the Calculator expression `add(2, 3)`, we would call the `Exp` constructor as follows: `Exp("add", [2, 3])`.

1.2 Evaluating Expressions

Our Calculator language is only concerned with two kinds of expressions: *numbers*, which are self-evaluating expressions, and *call expressions*, which involve an operator acting on some number of arguments, each of which are themselves expressions.

What does it mean for a number to be self-evaluating? If we evaluate an expression that is a number, the value of the expression (or the result returned by evaluating it) is that number.

What about evaluating call expressions? We follow this two-step process:

1. Evaluate the operands.
2. Apply the operator to the arguments (the values of the operands).

Using this two-step process, we can interpret any Calculator expression. The first step will be handled by a function called `calc_eval`, and the second step will be handled by a function called `calc_apply`. It is defined as:

```
def calc_eval(exp):
    if type(exp) in (int, float):  # If the expression is a number,
        return exp                  # Return the number.
    else:
        # Evaluate the operands.
        arguments = list(map(calc_eval, exp.operands))
        # Apply the function to the operands.
        return calc_apply(exp.operator, arguments)
```

As you can see, all we have done is follow the rules of evaluation outlined above. If the expression is a number, return it. Else, evaluate the operands and apply the operator on the evaluated operands.

How do we apply the operator? We use `calc_apply`:

```
def calc_apply(operator, args):
    if operator in ("add", "+"):
        return sum(args)
    if operator in ("sub", "-"):
        if len(args) == 0:
            raise TypeError(operator + " needs at least 1 arg")
        if len(args) == 1:
            return -args[0]
        return sum(args[0], [-args for args in args[1:]])
    if operator in ("mul", "*"):
        return reduce(mul, args, 1)
    if operator in ('div', '/'):
        if len(args) != 2:
            raise TypeError(operator + ' requires exactly 2 arguments')
        numer, denom = args
        return numer/denom
```

Depending on what the operator is, we can match it to a corresponding Python call. Each conditional in the function above corresponds to the application of one operator.

Notice that `calc_eval` deals with expressions, while `calc_apply` deals with values.

In general, when you want to add something to the calc language, you will want to edit one (or both) of `calc_eval` and `calc_apply`. Sometimes, it is possible to make changes in either function to get the same result, but it is important to consider which is more appropriate to edit. The general idea is:

- If you want to create a special rule for the order that we evaluate the operands of an expression, you will want to edit `calc_eval`.

- However, if you want to simply create a new function (with no special rules for the way it is evaluated), then it is generally better to edit `calc_apply` instead.

1. How would we construct the `Exp` object that corresponds to the expression

   `add(4, 5, mul(3, 2))`?

2. We want to allow Calc to perform exponentiation:

   ```
   >>> a = exp(2, 4)
   16
   >>> a = **(2, 4)
   ```

   Assuming that we have modified the parser to understand the new keywords `exp` and `**`, which of the two functions – `calc_eval` and `calc_apply` – would we modify to include this functionality, and how?

3. How many calls to `calc_eval` and `calc_apply` are produced by the expressions below?

   (a) `5`

   (b) `mul(5, 6)`

   (c) `mul(5, add(div(4, 1), 6))`

(d) `mul(sub(5, 1), add(4, 6))`

4. We want to add new functionality that would let us save values in variables:

```
calc> save(a, 5)
calc> mul(6, a)
30
```

Assuming that we have already modified the Calc parser to allow words to be operands, should we modify `calc_apply`, `calc_eval`, or both to add this new feature? Why, and how? (*Hint*: You may find it useful to keep a global dictionary.)

5. Louis Reasoner, while staying up all night in Soda, modified a line from his code for the Calc interpreter. In particular, he modified one line from `calc_eval`:

```
def calc_eval(exp):
    if type(exp) in (int, float):
        return exp
    if type(exp) == Exp:
        # The line below was changed.
        arguments = exp.operands
        return calc_apply(exp.operator, arguments)
```

Which of the following statements below will not be interpreted correctly?

1. `5`

2. `add(5, 3)`

3. `add(3, sub(2, 4))`

2 Scheme: Pairs and Lists

The basic data structure in Scheme is the *pair*. To construct a pair, we use the function **`cons`**:

```
STk> (cons 3 4)
(3 . 4)
```

Notice that Scheme represents a pair as the first element and the second element, separated with a dot. We can grab the first and the second elements of the list using the `car` and `cdr` function:

```
STk> (car (cons 3 4))
3
STk> (cdr (cons 3 4))
4
```

Pairs can be used to construct larger data structures, such as (recursive) lists. In fact, Scheme is a dialect of Lisp, which is a name derived from "List Processing". We can construct a list in many ways. The following expression should seem familiar to you from your adventures with IRLists and RLists.

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

What is a list? Well, a list is a pair whose first element is the first element of the list, and the second element is the rest of the list. We thus construct a list by first making a pair, the first element of which is the first element of the list. The second element of this pair is the rest of this list, so we construct another list. We build another pair whose first element is now the *second* element of the entire list. We keep adding elements to the list in this fashion until we reach the end, when we have no more elements to add and we use the empty list (represented as `'()` or `nil`).

Notice that above, Scheme did not print a pair back at you. This is because Scheme recognizes that you are building a list, and displays it as such. You can also build lists using the **`list`** operator:

```
STk> (list 1 2 3 4)
(1 2 3 4)
```

or the quote operator:

```
STk> '(1 2 3 4)
(1 2 3 4)
```

As with any other pair, we can still use the function `car` and `cdr`:

```
STk> (car '(1 2 3 4))
1
STk> (cdr '(1 2 3 4))
(2 3 4)
```

We can check if a list is empty using the **null?** function.

```
STk> (null? '(1))
#f
STk> (null? '())
#t
```

Note: `(car x)` is `2`, `(cdr x)` is the list `(3 4)`, and `(car (cdr x))` is 3! Well, it is a bit tiresome to write `(car (cdr x))` to get the second element of `x`. So Scheme, again the huggable lovable language that it is, provides a nifty shorthand: `(cadr x)`. This reads "cader", and means "take the `car` of the `cdr` of". Similarly, you can use `(caddr x)` (pronounced "caderder") to take the `car` of the `cdr` of the `cdr` of the list `x`, which is `4`. You can mix and match the `a` and `d` between the `c` and `r` to get the desired element of the list (up to four `a` or `d`).

You can also append two lists together. **append** takes in any number of lists and outputs a list containing those lists concatenated together. So, `(append (list 3 4) (list 5 6))` returns the list `(3 4 5 6)`.

1. Assume that we have typed in the following definitions:

   ```
   (define u (cons 2 3))
   (define v (list 2 3 4))
   (define w (cons 5 6))
   (define x (append v (list 2 5 6)))
   ```

   What do the following expressions evaluate to?

   (a) `(car u)`

   (b) `(cdr u)`

   (c) `(+ (cdr u) (car w))`

   (d) `(+ (car v) (cadr v))`

   (e) `(cons 3 v)`

We can also work with Scheme lists the same way we work with IRLists: treat the first and the rest of the list separately. For instance, here is the definition of a higher-order function **map** that takes in a function and returns a new list that results from applying that function on to each element of the list:

```
(define (map fn lst)
        (if (null? lst)
            '()
            (cons (fn (car lst))
                  (map fn (cdr lst)))))
```

1. Write the function `add-one`, which takes in a list and returns another list where 1 has been added to all of the elements.

```
(define (add-one lst)
```

2. Write the function `num-satisfies` that takes in a predicate function and a list, and returns the number of all the elements that satisfy the predicate.

```
(define (num-satisfies pred lst)
```

3. Write the function `filter` that takes in a predicate function (that returns `#t` or `#f`) and a list, and returns all the elements that satisfy the predicate.

```
(define (filter pred lst)
```

4. Write a function **`remove`** that takes in an element and a list, and returns a new list where the element has been removed.

```
(define (remove elem lst)
```