CS61A Notes – Week 2: Order of evaluation, recursion, procedures as data

The Substitution Model

We consider how Scheme evaluates an expression by using the substitution model. First, let's review how an expression is evaluated again:

1. Evaluate all subexpressions of the combination (including the procedure and arguments) so that we know exactly what they are.
2. Apply the evaluated procedure to the evaluated arguments that we obtained in the previous step.

To perform step 2 we evaluate the body of the procedure after we replace the formal parameters with the given arguments. Let's dive right in. First, a few functions to play with:

```
(define (double x) (+ x x))
(define (square y) (* y y))
(define (f z) (+ (square (double z)) 1))
```

Now, what happens when we do `(f (+ 2 1))`? Here's how the substitution model wants us to think:

1. Let's evaluate the expression: `f` is a `lambda`; what's `(+ 2 1)`?
   `(f `**`3`**`)`
2. Now, let's take the body of `f`, and *substitute* `3` for `z`:
   `(+ (square (double 3)) 1)`
3. Now, we evaluate the expression: `+` is `+`, `1` is `1`, but what is `(square (double 3))`? Well, `square` is a procedure; what is `(double 3)`? Well, `double` is a procedure, and `3` is `3`! So now that we've evaluated everything, we can start applying the procedures. First we do `double` by substituting the body in again:
   `(+ (square `**`(+ 3 3)`**`) 1)`
4. Now, what's `(+ 3 3)`?
   `(+ (square `**`6`**`) 1)`
5. How do we apply `square`? Let's substitute in the body of `square`:
   `(+ `**`(* 6 6)`**` 1)`
6. So what's `(* 6 6)`?
   `(+ `**`36`**` 1)`
7. And what's `(+ 36 1)`?
   `37`

If the above seems a little obvious, it's because it is – the substitution model is the most intuitive way to think about expression evaluation. Note, however, that this is not how Scheme actually works, and as we'll find out later in the course, the substitution model eventually will become inadequate for evaluating our expressions – or, more specifically, when we introduce assignments. But for now, when we're still in the realm of functional programming, the substitution model will serve us nicely.

Applicative vs. Normal Order

The substitution model example presented above was done using "applicative order", where Scheme first evaluates completely all the procedures and arguments before executing the function call. Another way to do this – in "normal order" – is to expand out complex expressions involving defined procedures into one involving only primitive operators and self-evaluating values, and then perform the evaluation. In other words, we *defer* evaluation of the arguments; we substitute the *unevaluated* expression (as opposed to evaluated expression for applicative order) into the body of a procedure.

So for the same problem `(f (+ 2 1))`:

1. We'll expand `f` into its body first, leaving `(+ 2 1)` unevaluated:
   `(+ (square (double (+ 2 1))) 1)`
2. Now, we want to expand `square`; let's write in the body of `square`, and substitute `(double (+ 2 1))` for `y`, leaving `(double (+ 2 1))` unevaluated:
   `(+ `**`(* (double (+ 2 1)) (double (+ 2 1)))`**` 1)`
3. Now, we want to expand `double`; let's write the body of `double`, and substitute `(+ 2 1)` for `x`:
   `(+ (* `**`(+ (+ 2 1) (+ 2 1)) (+ (+ 2 1) (+ 2 1))`**`) 1)`
4. Finally we've expanded everything to just primitive operators and self-evaluating values. We'll do these one by one…
   ```
   (+ (* (+ 3 (+ 2 1)) (+ (+ 2 1) (+ 2 1))) 1)
   (+ (* (+ 3 3) (+ (+ 2 1) (+ 2 1))) 1)
   (+ (* (+ 3 3) (+ 3 (+ 2 1))) 1)
   (+ (* (+ 3 3) (+ 3 3)) 1)
   (+ (* 6 (+ 3 3)) 1)
   (+ (* 6 6) 1)
   (+ 36 1)
   37
   ```

Well okay, that seems harmless enough. It's just two ways of thinking about the problem: for applicative order, we evaluate completely all subexpressions first before we apply the current procedure. For normal order, we expand all the defined procedures into their bodies until everything is expressed in terms of primitive operations and self-evaluating values; then we find the answer.

So what's the difference? Some of you will raise concerns about efficiency (obviously, in this case, normal order causes us to perform `(+ 2 1)` four times, whereas we only performed the same calculation once for applicative order!). But in terms of correctness, the two are the same. That is, **in *functional programming*, applicative and normal order of evaluation will *always* return the same answer!**

QUESTIONS

In Class:

1. Evaluate this expression using both applicative and normal order: `(square (random x))`. Will you get the same result from both? Why or why not?

2. Consider a magical function `count` that takes in no arguments, and each time it is invoked, it returns 1 more than it did before, starting with 1. Therefore, `(+ (count) (count))` will return 3. Evaluate `(square (square (count)))` with both applicative and normal order; explain your result.

Extra Practice:

3. Above, applicative order was more efficient. Define a procedure where normal order is more efficient.

Yoshimi Battles the Pink Recursive Robots

A "recursive procedure" is one that calls itself in its body. The classic example is factorial:

```
(define (fact n)
    (if (= n 0)
```

```
            1
            (* n (fact (- n 1)))))
```

Note that, in order to calculate the factorial of n, we tried to first calculate the factorial of `(- n 1)`. This is because of the observation that `5! = 5 * 4!`, and that, if we know what `4!` is, we could find out what `5!` is. If this makes you a bit suspicious, it's okay – it takes a little getting used to. The trick is to use the procedure *as if it is already correctly defined*. This will become easier with practice. The mantra to repeat to yourself is:

TRUST THE RECURSION!

QUESTIONS

In Class:

1. **Write a procedure `(expt base power)` which implements the exponents function. For example, `(expt 3 2)` returns 9, and `(expt 2 3)` returns 8.**

2. **There is something called a "falling factorial". `(falling n k)` means that `k` consecutive numbers should be multiplied together, starting from `n` and working downward. For example, `(falling 7 3)` means `7 * 6 * 5`. Write the procedure `falling` that generates an iterative process.**

Extra Practice:

3. **Define a procedure `subsent` that takes in a sentence and a parameter `i`, and returns a sentence with elements starting from position `i` to the end. The first element has `i = 0`. In other words,**

```
(subsent '(6 4 2 7 5 8) 3) => (7 5 8)
```

4. **Write a version of `(expt base power)` that works with negative powers as well.**

5. **Define a procedure `sum-of-sents` that takes in two sentences and outputs a sentence containing the sum of respective elements from both sentences. The sentences do not have to be the same size!**

```
(sum-of-sents '(1 2 3) '(6 3 9)) => (7 5 12)
(sum-of-sents '(1 2 3 4 5) '(8 9)) => (9 11 3 4 5)
```

What in the World is `lambda`?

No, here, `lambda` is not a chemistry term, nor does it refer to a brilliant computer game that spawned an overrated mod. `lambda`, in Scheme, means "a procedure". The syntax:

```
(lambda ([formal parameters]) [body of the procedure])
```

So let's use a rather silly example:

```
(lambda (x) (* x x))
```

Read: a procedure that takes in a single argument `x`, and returns a value that is that argument `x` multiplied by itself. When you type this into Scheme, the expression will evaluate to something like:

```
#[closure arglist=(x) 14da38]
```

This is simply how Scheme prints out a "procedure". Since `lambda` just gives us a procedure, we can use it the way we always use procedures – as the first item of a function call. So let's try squaring `3`:

```
((lambda (x) (* x x)) 6)
```

Read that carefully and understand what we just did. Our statement is a function call, whose function is "a procedure that takes in a single argument, and returns a value that is that argument multiplied by itself", and whose argument is the number `6`. Compare with `(+ 2 3)`; in that case, Scheme evaluates the first element – some procedure – and passes in the arguments `2` and `3`. In the same way, Scheme evaluates the first element of our statement – some procedure – and passes in the argument `6`.

A lot of you probably recognized the above `lambda` statement as our beloved "`square`". Of course, it's rather annoying to have to type that `lambda` thing every time we want to square something. So we can bind it to a name of our desire. Oh, let's just be spontaneous, and use the name "`square`":

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

Note the similarities to `(define pi 3.1415926)`! For `pi`, we told Scheme to "bind to the symbol '`pi`' the value of the expression `3.1415926` (which is self-evaluating)". For `square`, we told Scheme to "bind to the symbol '`square`' the value of the `lambda` expression (which happens to be a procedure)". The two cases are not so different after all. If you type `square` into the Scheme prompt, it would print out something like this again:

```
#[closure arglist=(x) 14da38]
```

As we said before, this is how Scheme prints out a procedure. Note that this is not an error! It's simply what the symbol "`square`" evaluates to – a procedure. It's as natural as typing in "`pi`" and getting `3.1415926` back (if you've already defined `pi`, of course).

It's still a little annoying to type that `lambda` though, so we sugar-coat it a bit with some "syntactic sugar" to make it easier to read:

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

Ah, now that's what we're used to. The two ways of defining `square` are exactly the same to Scheme; the latter is just easier on the eyes, and it's probably the one you're more familiar with. Why bother with "`lambda`" at all, then? That is a question we will eventually answer, with vengeance.

QUESTIONS: What do the following evaluate to?

`(lambda (x) (* x 2))`

`((lambda (a) (a 3)) (lambda (z) (* z z)))`

First-class Procedures!

In programming languages, something is **first-class** if it doesn't chew with its mouth open. Also, as *SICP* points out, it can be bound to variables, passed as arguments to procedures, returned as results of procedures and used in data structures. Obviously we've seen that numbers are first-class – we've been passing them in and out of procedures since the beginning of time (which is about last week ago), and have been putting them in data structures – sentences – since the dinosaurs went extinct (also last week).

What is surprising – and exceedingly powerful – about Scheme, is that **procedures are also awarded first-class status**. That is, procedures are treated just like any other values! This is one of the more exotic features of LISP (compared to, say C or Java, where passing functions and methods around as arguments require much more work). And as you'll see, it's also one of the coolest.

Procedures as Arguments

A procedure which takes other procedures as arguments is called a **higher-order procedure**. You've already seen a few examples, in the lecture notes, so I won't point them out again; let's work on something else instead. Suppose we'd like to square or double every word in the sentence:

```
(define (square-every-word sent)
    (if (empty? sent)
        '()
        (se (* (first sent) (first sent)) (square-every-word (bf sent)))))

(define (double-every-word sent)
    (if (empty? sent)
        '()
        (se (* 2 (first sent)) (double-every-word (bf sent)))))
```

Note that the only thing different about `square-every-word` and `double-every-word` is just what we do to `(first sent)`! Wouldn't it be nice to generalize procedures of this form into something more convenient? When we pass in the sentence, couldn't we specify, also, what we want to do to each word of the sentence?

To do that, we can define a higher-order procedure called `every`. `every` takes in the procedure you want to apply to each element *as an argument*, and applies it indiscriminately. So to write `square-every-word`, you can simply do:

```
(define (square-every-word sent) (every (lambda(x) (* x x)) sent))
```

Now isn't that nice. Nicer still: the implementation of `every` is left as a homework exercise! You should be feeling all warm and fuzzy now.

The secret to working with procedures as values is to **keep the domain and range of procedures straight**! Keep careful track of what each procedure is supposed to take in and return, and you will be fine. Let's try a few:

QUESTIONS

In Class:

1. **What does this guy evaluate to?**
   `((lambda (x) (x x)) (lambda (y) 4))`

2. **What about his new best friend?**
   `((lambda (y z) (z y)) * (lambda (a) (a 3 5)))`

`Extra Practice:`

3. **Write a procedure, `foo`, that, given the call below, will evaluate to `10`.**
   `((foo foo foo) foo 10)`

4. **Write a procedure, `bar`, that, given the call below, will evaluate to `10` as well.**
   `(bar (bar (bar 10 bar) bar) bar)`

Procedures as Return Values

The problem is often: write a procedure that, given _____, **return a procedure** that _____. The keywords – conveniently boldfaced – are that your procedure is supposed to return a procedure. This is often done through a call to `lambda`:

```
(define (my-wicked-procedure blah) (lambda(x y z) (...)))
```

Note that the above procedure, when called, will return a **procedure** that will take in three arguments. That is the common form for such problems (of course, never rely on these "common forms" as they'll just work against you on midterms!).

QUESTIONS

In lecture, you were introduced to the procedure `keep`, which takes in a predicate procedure and a sentence, and throws away all words of the sentence that doesn't satisfy the predicate. The code for `keep` was:

```
(define (keep pred? sent)
    (cond ((empty? sent) '())
          ((pred? (first sent))
           (sentence (first sent) (keep pred? (bf sent))))
          (else (keep pred? (bf sent)))))
```

Recall that Brian said to `keep` numbers less than 6, this wouldn't work: `(keep (< 6) '(4 5 6 7 8))`

1. **Why doesn't this work?**

2. **Of course, this being Berkeley, and us being rebels, we're going to promptly prove the authority figure – the Professor himself – wrong. And just like some rebels, we'll do so by cheating. Let's do a simpler version; suppose we'd like this to do what we intended:**
   `(keep (lessthan 6) '(4 5 6 7 8))`

   Define procedure `lessthan` to make this legal.

3. **Now, how would we go about making this legal?**
   `(keep (< 6) '(4 5 6 7 8))`

<u>Common Higher Order Functions:</u>

```
Every:  (apply a function to each element of a sentence)

      (every <function> <sent>)

      > (every square '(1 2 3 4))
      > (1 4 9 16)

Keep:  (filter out the elements of a sentence that satisfy a given predicate)

      (keep <predicate> <sent>)

      > (keep even? '(1 2 3 4))
      > (2 4)

Accumulate:  (reduce a sentence to a single element using an operator and initial value)

      (accumulate <operator> <initial value> <sent>)

      > (accumulate + 0 '(1 2 3 4))
      > 10
```

Secrets to Success in CS61A

- Ask questions. When you encounter something you don't know, ask. That's what we're here for. (Though this is not to say you should raise your hand impulsively; some usage of the brain first is preferred.) You're going to see a lot of challenging stuff in this class, and you can always come to us for help.
- Go to office hours. Office hours give you time with the professor or TAs by themselves, and you'll be able to get some (nearly) one-on-one instruction to clear up confusion. You're NOT intruding; the professors LIKE to teach, and the TAs…well, the TAs get paid. Remember that, if you cannot make office hours, you can always make separate appointments with us!
- Do the readings (on time!). There's a reason why they're assigned. And it's not because we're evil; that's only partially true.
- Do all the homeworks. Their being graded on effort is not a license to screw around. We don't give many homework problems, but those we do give are challenging, time-consuming, but very rewarding as well.
- Do all the labwork. Most of them are simple and take no more than an hour or two after you're used to using emacs and Scheme. This is a great time to get acquainted with new material. If you don't finish, work on it at home, and come to office hours if you need more guidance!
- Study in groups. Again, this class is not trivial; you might feel overwhelmed going at it alone. Work with someone, either on homework, on lab, or for midterms (as long as you don't violate the cheating policy!).