Lecture 2: Recursive and Iterative Processes

There are practice exercises at the end of this file. I highly recommend spending about 20 minutes on **Pow**! Just one problem can be extremely helpful. Then do the rest if you have the energy to do it.

Links

- Lecture 1 (last week) notes: https://docs.google.com/document/d/1789LJjFQg3paf-2ekXXZMIgoTRoYNtQjtMzdZGrralw/edit#heading=h.eursqe10bbcz
- The book (reference chapter 1.2 for this week): https://mitp-content-server.mit.edu/books/content/sectbyfn/books_pres_0/6515/sicp.zip/full-text/book/book-Z-H-4.html#%_toc_start
- **Lecture 2 (this week) slides:** https://docs.google.com/presentation/d/1lnCZhQiBmFmoJuTQzjvVb7Q2WgqMPQb2ay85TwCYB_o/edit?usp=sharing

Recap

- **Warmup question**: Compute the following. (SICP Exercise 1.1)
  - `10`
  - `(+ 5 3 4)`
  - `(- 9 1)`
  - `(/ 6 2)`
  - `(+ (* 2 4) (- 4 6))`
  - `(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))`

Updated 2023-03-04 4:55pm

    -
      ```
      (* (cond ((> a b) a)
               ((< a b) b)
               (else -1))
         (+ a 1))
      ```
- Last time we motivated the subject by talking about:
    - A pretty cool trick for finding square roots.
    - Generalizing the trick to find the fixed point of the cosine function, i.e. solve cos(x) = x.
    - The distinction between declarative knowledge versus imperative knowledge, i.e. procedural explanations that actually tell us how to do something.
- We then introduced Scheme's syntax by talking about three elements of a powerful language.
    - Primitives
        - Numbers
    - Means of combinations
        - Function application using prefix notation
        - Cond form for creating conditionals
    - Means of abstraction
        - Defining a name for a value
        - Defining a procedure

Expressions and the substitution model

- We're gonna define some formal terminology for things we have already seen in Scheme.
- In Scheme, everything we write is an "**Expression**." There are two kinds of expressions.
    - **Atomic expression** can be either a "symbol" or a "literal."
        - **Symbols** are names for variables or procedures. Every time Scheme counters a symbol, it evaluates the symbol by looking it up on a table of some sorts.
            - Symbols usually consist of alphabetic characters and dashes.
        - **Constant literals** include numbers, strings, and booleans.
            - Note that there is a philosophical distinction between the "representation" and the actual abstract object. For example, if you write 3, it's just a representation of the actual abstract concept of 3.
            - When you write something like (+ 3 4), what Scheme is really doing is giving you the abstract operator plus, three, and four,

        then performing that addition. It gets the number seven, then tries to represent that with the symbol 7.
    - When you write something like + into Scheme, Scheme wants to tell you that this is a procedure (in some abstract sense), so it will display #:<procedure>.
- **S-expression** is a list of expressions (which can themselves be atomic or S-expr).
    - In the **general case**, S-expressions are **function applications**. First, Scheme evaluates the expressions in the operator and operand positions. Then, it checks whether the supposed operator is indeed an operator and whether it takes the correct number of arguments. Finally, it applies the operator to the operands, replacing the entire S-expression with the final result.
        - Example: **sum-of-squares**
          ```
          (define (sq x)
            (* x x))
          (define (sos x y)
            (+ (sq x) (sq y))
          (sos 3 4)
          -> (+ (sq 3) (sq 4))
          -> (+ (* 3 3) (sq 4))
          -> (+ 9 (sq 4))
          -> (+ 9 (* 4 4))
          -> (+ 9 16)
          -> 26
          ```
    - Other cases are called "**special forms**." Those have their own special evaluation rules. So far we have seen *define* and *cond*.
    - **Take a look at the slides for more details here.**
        - Note that in Racket you can use either brackets [] or parentheses (). We commonly use brackets to denote clauses in conditionals and name-value pairs in lets.
    - These have to be special forms because if we apply the general rule, some terms that should not be evaluated will get evaluated. (For example, the word "define" cannot be looked up as a variable. Only one branch of a cond expression should be executed.)
    - If *if* was not a special form, can you come up with a program that is potentially problematic? (Hint: let's say you want to divide two numbers, but only when the divisor is not zero.)

Updated 2023-03-04 4:55pm

- All of this gives us a very concrete set of steps to take whenever we see an expression.
  - First, check if the expression is a constant literal. If it is, then it is what it is.
  - If the expression is a symbol, then look up what the name corresponds to.
  - If the expression is an S-expression (i.e. a list), see what the first symbol is. If it refers to a special form, execute the special form.
  - Otherwise, compute the operator and all the operands then apply the operator.
- This may or may not be how Scheme runs under the hood. Either way, it is irrelevant. What matters now is that we have a "good enough" model that will let us gain more understanding of the language. We will refine the model when we introduce new features in Scheme (actually, in week 5 or week 6).
- Let's run through this example.

  -
    ```
    (define (a-plus-abs-b a b)
      (if (>= b 0)
          (+ a b)
          (- a b)))
    (define (a-plus-abs-b-again a b)
      ((if (>= b 0) + -) a b))
    ```
  - What happens when I execute `(a-plus-b 5 -3)`?
  - What about `(a-plus-b-again 5 -3)`?

Shapes of processes

- Suppose all we have is just constants and two operations *inc* and *dec*. How can we define *+*? Let's assume both arguments are non-negative numbers.
  - This might feel like a bit of a contrived example, but one thing to keep in mind is that you never really know what is available in a language and what is not. For each language, there is a documentation that lists what you have that you could probably consult. But when something doesn't exist, you can build your own by combining and abstracting the other primitives that are available!
    - For example, in a normal situation, you're likely going to define *inc* and *dec* in terms of + and -, but let's just do it the other way around for demonstration purposes.
  - The first way:

    -
      ```
      (define (+ x y)
        (if (= x 0)
            y
            (+ (dec x) (inc y))))
      ```
  - The second way:

        -
          ```
          (define (+ x y)
            (if (= x 0)
                y
                (inc (+ (dec x) y))))
          ```
- **Important:** When you don't understand something in programming, **mechanically** work through an example until you get a sense of what it does. Be as granular as you need to be.
- We walk through how the process of evaluating `(+ 5 3)` looks like.
    - The first way:

        -
          ```
          (+ 5 3)
          -> (+ 4 4)
          -> (+ 3 5)
          -> (+ 2 6)
          -> (+ 1 7)
          -> (+ 0 8)
          -> 8
          ```
    - The second way:

        -
          ```
          (+ 5 3)
          -> (inc (+ 4 3))
          -> (inc (inc (+ 3 3)))
          -> (inc (inc (inc (+ 2 3))))
          -> (inc (inc (inc (inc (+ 1 3)))))
          -> (inc (inc (inc (inc (inc (+ 0 3))))))
          -> (inc (inc (inc (inc (inc 3)))))
          -> (inc (inc (inc (inc 4))))
          -> (inc (inc (inc 5)))
          -> (inc (inc 6))
          -> (inc 7)
          -> 8
          ```
- Note that we get two different shapes. Both methods use the same amount of time. The first method uses constant space while the second method uses space linear in the value of `x`. Scheme has to keep track of a lot to be able to perform the computation: it has to perform the inner computations first before performing the outer operation.
    - In programming jargon, we call this *O(1)* vs *O(x)*.
- Both procedures are written recursively, but they generate different processes. The first way produces an **iterative process**. The second way produces a **recursive process**. Iterative process has all the state data it needs in the arguments. Recursive process requires additional book-keeping. We could also get a shape that expands exponentially. See the Fibonacci function.

Updated 2023-03-04 4:55pm

  -
    ```
    (define (fib n)
      (if (<= n 1)
          n
          (+ (fib (- n 1))
             (fib (- n 2)))))
    ```
- If you walk through how (fib 5) expands, you will see that it is extremely inefficient. This is because there is a lot of redundant work. To compute (fib 5), you need to compute (fib 4) and (fib 3). To compute (fib 4), you need to compute (fib 3) and (fib 2). Note that the answer to (fib 3) here should be shared with (fib 3) from earlier, if we want this to be efficient.
- To make things efficient, we look into human intuition to figure out how we normally compute the nth Fibonacci number.
  - Let's say you want to compute fib(10). You probably list out fib(0) = 0 and fib(1) = 1 first, then you keep building up the list by adding the two previous numbers. i.e. fib(2)=1, fib(3)=2, fib(4)=3, fib(5)=5, fib(6)=8, … This should take linear time.
- We can represent this iterative process similarly to how we do addition using the first way. To do this, you have to figure out what "states" you want to remember as you keep iterating through your process. For Fibonacci, you just need the current value of n, the goal value n, and the two most recent numbers.
- Those states can be represented by the operands. So, we can write our Fibonacci procedure like this.

  -
    ```
    (define (iter current-n goal-n recent second-recent)
      (if (= current-n goal-n)
          (+ recent second-recent)
          (iter (+ current-n 1)
                goal-n
                (+ recent second-recent)
                recent)))
    (define (fib n)
      (if (<= n 1)
          n
          (iter 2 n 1 0)))
    ```
- Again. To get a good sense of how this works, try this out mechanically!

  -
    ```
    (fib 10)
    -> (iter 2 10 1 0)
    -> (iter 3 10 1 1)
    -> (iter 4 10 2 1)
    -> (iter 5 10 3 2)
    -> (iter 6 10 5 3)
    ```

Updated 2023-03-04 4:55pm

```
-> (iter 7 10 8 5)
-> (iter 8 10 13 8)
-> (iter 9 10 21 13)
-> (iter 10 10 34 21)
-> (+ 34 21)
-> 55
```

- As you can see, we keep track of just about enough information to be able to keep on iterating until we reach the end. It might be a bit confusing to see what number goes into what position, but with practice you will be able to write efficient iterative procedures like this.
- Next week we will talk a bit more about recursion then go into higher-order functions if we have time.

Practice exercises

- This lecture contains a very fundamental idea that takes a lot of time to grasp. If you have time, I highly recommend that you work through the exercises below. Please do **Pow** at the very least! (Give yourself 20-30 minutes to work on it.)
- If you get stuck, feel free to email me for help! If you solve a problem, consider sharing your solution on https://pastebin.com/ and pasting the link here!
- **Pow**: Implement a recursive procedure *pow* that computes *a* to the power of *b*.
    - For example, *(pow 3 5)* should output 243.
    - Fill in the blank:
      ```
      (define (pow a b)
        (if (= b 0)
            1
            ____________________))  ;hint: we want b to eventually reach 0!
      ```
    - Now, check that your answer is correct by expanding *(pow 3 5)* by hand. Also check with DrRacket if possible.
- **Pow Iterative**: Make it iterative.
    - Observe the template below. We are defining pow in terms of pow-iter, which keeps track of four pieces of information: a, current value of b, goal b, and the current product.
    - Fill in the blank:
      ```
      (define (pow-iter a current-b goal-b current-answer)
        (if (= current-b goal-b)
            ___________________
            ______________________________________________________________)
      (define (pow a b)
        (pow-iter a 0 b 1))
      ```

Updated 2023-03-04 4:55pm

- **Factorial**: Implement a recursive procedure *fact* that computes *n!*.
    - Recall *n!* is defined as *n*(n-1)*(n-2)*…*1*.
    - Template:
      ```
      (define (fact n)
        …)  ; hint: should feel very similar to Pow!
      ```
- **Factorial Iterative**: This is also somewhat similar to Pow, but instead of counting up from 0 to n, let's count down from n to 0 instead.
    - Template:
      ```
      (define (fact n)
        (fact-iter n 1)
      (define (fact-iter n ans)
        (if (= n 0)
            ans
            ________________________________))
      ```
- **Prime counting**: Suppose you have a procedure *(prime? n)* which returns #t if n is a prime, #f otherwise. How would you count the number of primes from 1 to n?
    - For example, we want *(count-primes 100)* to output 25 because there are a total of 25 prime numbers between 1 and 100.
    - Template:
      ```
      (define (count-primes n)
         …)
      ```
    - This can be defined recursively or iteratively. Try whichever one you want!
- **Prime checking (Challenge)**: How would you go about implementing *prime?* yourself? Recall the definition of a prime number: *n* is a prime number if it has exactly two divisors, *1* and *n*, and nothing else.

Updated 2023-03-04 4:55pm