Week 7: Developing a Lisp or Clojure Function

Coping with confusing syntax
Incremental development
Final (production) quality and documentation

(defn glop [x y] )

An obvious problem

- Code written in functional programming languages (Lisp, Clojure, etc.) is often hard to read, especially before we've become accustomed to the idiom.

- It's too easy to get stuck in confusing debugging situations, where we're not sure exactly what happened or how to recover.

- Therefore, we need to adopt a cautious, systematic, incremental approach to composing and validating our functions.
Don't worry

- Everyone experiences start-up difficulties with a first functional programming language.

- Let's look at an example we're all familiar with (from assignment #1): the integer power function.

The algorithm

- We recall from assignment #1 (in C, C++, C#, or Java) and from elementary algebra that for integer $n$
  
  $$a^{2n} = (a^n)^2$$
  $$a^{2n+1} = a \times a^{2n}$$

  suggesting a recursive computation.

- And also base cases:
  $$a^0 = 1$$
  $$a^{-n} = 1 / a^n$$
Getting started

- Let's start with the base cases: \textit{Why?}

user=>
  (defn power [x n]
   (cond (= n 0) 1
       (= n 1) x))
user=> nil

- We can test this right away \textit{Why?}

Validate the base cases

- 

user=> (power 44 0)
  1
user=> (power 66 1)
  66

- They work!
Enhancing readability
- Let's align some matching parentheses:

  ```clojure
  (defn power [x n]
      (cond (= n 0) 1
        (= n 1) x
      )
  )
  user=> nil

- This may help as we add more levels

Adding another case
- Taking care of odd n

  ```clojure
  (defn power [x n]
      (cond (= n 0) 1
        (= n 1) x
        (odd? n) (* x (power x (- n 1)))
      )
  )
  user=> nil

- Can we test more cases now?
- Should we align the new right parentheses?
What's left?

- Taking care of even n

```clojure
user=>
   (defn power [x n]
      (cond (= n 0) 1
            (= n 1) x
            (odd? n) (* x (power x (- n 1)))
            true    (expression for n even)
      )
   )
user=> nil
```

What goes there?

How's this?

```clojure
user=>
   (defn power [x n]
      (cond (= n 0) 1
            (= n 1) x
            (odd? n) (* x (power x (- n 1)))
            true    (* (power x (/ n 2))
                       (power x (/ n 2)))
      )
   )
user=> nil
```

What's wrong with that?
Now let's run some tests

user=> (power 2.0 3)
8.0
user=> (power 2.0 15)
32768.0
user=> (power 2.0 24)
1.6777216E7
user=> (power 5.0 6)
15625.0
user=> (power 1.06 30)
5.743491172913265

■ Looking good!

So far, so good. but . . .

■ Oh, oh!

user=> power 2.0 -3)
StackOverflowError clojure.lang.Numbers.minus
(Numbers.java:1713)

■ We forgot about negative integer powers!
Handling negative powers

- We insert code for that case (another recursive invocation)

```
user=>
    (defn power [x n]
        (cond (= n 0) 1
            (= n 1) x
            (< n 0)  (power (/ x) (- 0 n))
            (odd? n) (* x (power x (- n 1)))
            true    (* (power x (/ n 2))
                     (power x (/ n 2)))
    )
)
user=> nil
```

- Now we just need to test it.

Final (?) testing

- Verify that negative exponents work:

```
user=> (power 2.0 -3)
  0.125
```

- Now we should re-run all our tests to make sure we didn't spoil anything.

- Hooray! They still work.

- But we're still not satisfied

  What's wrong?
Unnecessary repetition

- The even-exponent case invokes the function twice for the same arguments!

true \( \ast (\text{power } x \ (\div n 2)) \ (\text{power } x \ (\div n 2)) \) 

- A sophisticated optimizing compiler might recognize that and avoid the inefficient second invocation, but:
  - We can't depend on that
  - Besides, the repetition makes the code confusing and harder to understand.

- Let's correct that problem.

One solution

- A read-only helper function

```clojure
(defn square \[x\] \(* \ x \ x\))

(defn power \[x n\]
  (cond (= n 0)  1
        (= n 1)  x
        (< n 0)  (power \(/ x\) (\- 0 n))
        (odd? n) \(* x (power x (\- n 1))\)
        true    (square (power x \(/ n 2\)))
  )
)
```

- After we learn more Clojure we may think of a better way, but this is OK for now.
So, are we done?

No! What's still wrong with this?

```clojure
(defn power [x n]
  (cond (= n 0) 1
        (= n 1) x
        (< n 0) (power (/ x) (- 0 n))
        (odd? n) (* x (power x (- n 1)))
        true (square (power x (/ n 2)))
  )
)
```

Should we submit that to a module library for other programmers to use?

Not ready for production status!

- Our function needs commentary
  - Some experts ("agilists") claim that code should be so clear that it doesn't need explanation.
  - They're wrong! Some code in some programming languages may be obvious, and we needn't (shouldn't) document it:
    ```clojure
    ++tblx; // Increment the table index
    ```
  - But a potential user or maintenance programmer shouldn't have to decipher complicated code to figure out what the original programmer intended.

- Let's annotate our final version.
One final detail

- We learned in algebra class that for non-zero \( n \):
  
  \[
  0^n = 0 \\
  n^0 = 1
  \]

- But what about \( 0^0 \)? That's undefined!

- What should the function do?

- Raising an exception is complicated and expensive for a condition that will probably never occur. Let's just say the result is "undefined".

The final version

```clojure
(defn power [x n]
  ; Real number to integer power
  ; Result is undefined for x & n both 0
  (cond (= n 0) 1 ; Base
   (= n 1) x ; cases
   (< n 0) (power (/ x) (- 0 n)) ; Negative exponent
   (odd? n) (* x (power x (- n 1))) ; Odd exponent
   true (square (power x (/ n 2))) ; Even exponent
  )
)

Is that clear now?
```
Lessons learned

- Build (code & unit test) functionality incrementally
  - Then when something doesn't work, it's very likely a problem with the *most recent* incremental change.

- Document
  - as you go if it helps
  - at the end if you haven't done it already