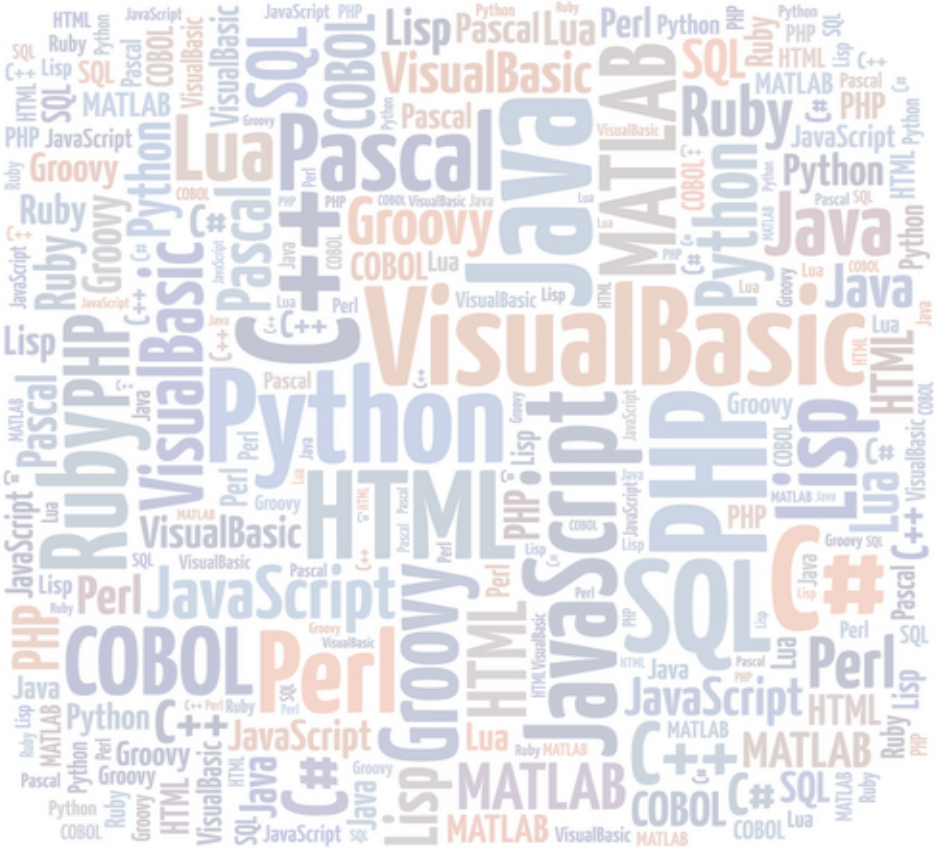
COMP 6411: Comparative Programming Languages

Clojure Programs



Introduction

- Absence of loops
- Clojure recursion
- Initializing variables using recursions
- Use of recur
- Libraries and namespace
- Hiding functions
- Importing Clojure and Java libraries
- Instantiating Java classes

Processing data structures

- If at all possible, we want use the core Clojure functions to operate on basic data structures
  - E.g., map, reduce, filter
- These are easy, intuitive, and they are guaranteed to work properly.
- That being said, there will always be times when you need to provide some sort of customized functionality.
- As noted, there are no FOR/WHILE loops in Clojure.
- So we need another mechanism to iterate over the data in the core structures.

Recursion

- When loops are not available, recursion becomes the mechanism by which iterative processing is performed.
- Most imperative languages, including Java and C, include mechanisms for recursion.
- But since they are never actually required, many programmers will opt for loops
  - Even if recursion would be a simpler solution.
- In Clojure you do not have that option.
  - You must use recursion.

Recursion…cont'd

- Recursion is relatively simply to describe
  - Though not always easy to implement
- To summarize, recursion has 3 basic features
  1. A function calls itself repeatedly
  2. The recursive invocations terminate when a programmer-specified condition is met
  3. The recursion then "unwinds" back to the first invocation of the function.
- Note that if the terminating condition does not occur, the program will crash with an "out of stack space error"
  - By default, each recursive invocation creates another stack frame until the terminating condition is met

Recursion…cont'd

- In Clojure, no data is shared inside the function invocations.
- So any "sharing" must occur *between* the function invocations
  - In other words, sharing is done via the arguments and return values.
- How would we, for example, reverse the values in a vector of ints?

```
(defn flip
  [numbers]
  (if (empty? numbers)
    []
    (conj (flip (rest numbers)) (first numbers) )))
```

simple.clj

A little more recursion

- We can of course combine recursion with multi-arity methods in order get more sophisticated results.
- This might be necessary when one needs to initialize a value in just one instance of a function invocation.
- For example, how would you sum up a list of values in a vector, as in the following C sample?

```
// iterative summation function

int sum(int *items, int size){
   int sum = 0;  // this is only intialized once
   int i;
   for(i = 0; i < size; i++){
     sum += items[i];
   }
   return sum;
}
```

Using recur

- Note that basic recursion can be implemented simply by making a recursive call.
- However, in cases where the recursion is the last expression in the function, it is recommended that you use the `recur` function.
- Clojure does not directly support "tail-call" optimization.
  - So a large number of stack frames can be created
- By using recur, tail-call recursions are transformed into something resembling a while loop
  - This uses a constant amount of stack space and is far more memory efficient.

recurz.clj

Libraries and namespaces

- In general, the Clojure mechanisms for defining and using “packages” of code follow the Java model.
- In practice, there are a number of special options but our focus will be on the core tools
- To start, note that Clojure will associate your code with a default namespace called “user”
- This works but we would generally like to provide proper names for our units of code.

namespaces

- In Clojure, we refer to a collection of functions as a library.
- Each library is associated with its own *namespace*
  - Conceptually similar to a Java package.
- Like Java (and Python), we will import namespaces into our the current Clojure file (or namespace) so that the associated functions can be used.
  - Note that a library will only be processed once
    - We do no need header guards like C.

Namespace…cont'd

- Namespaces are defined at the top of your closure library file.
- For a library, we can simply use the "ns" syntax to define the namespace, as indicated below.
  - Here, we are defining a myproj.foo namespace
  - Physically, foo will be mapped to a myproj/foo path
- Note that for large projects, we have to be concerned with the Java **classpath**, as namespace locations are relative to this set of paths
  - In practice, large Clojure projects are built with a framework called **leiningen**, which handles this for you.

```
(ns myproj.foo)

…
foo code
```

Privacy

- Clojure has no `public` or `private` keywords.
  - So all functions and vars are public by default
  - Because of its emphasis on data immutability, this is perhaps less important.
- Still, if you want to indicate that a function is not part of the public API, there are a couple of ways to do this.
  - use `defn-` instead of `defn` when defining a function
    - This works for functions only
  - use `^:private` as a metadata tag for the compiler
    - This also works for data

```
(defn- boo [y] (* (* y y) y))

(def ^:private boo [y] (* (* y y) y))
```

Importing other libraries

- Of course, we will need to import other libs into the current file/namespace
- To do this, we will augment the current namespace definition.
- Specifically, we will indicate that other namespaces are *required* by the current namespace.
  - We can then use the fully qualified path of the imported namespace to invoke the function
  - We use a "/" to separate the namespace path from the function

```
(ns test.foo
  (:require test.bar))  ; a library of functions in test/bar

(test.bar/myfunc x)     ; myfunc is a funcion in test/bar
```

Namespaces…cont'd

- We can also use additional specifications for the import.
- For example, we can abbreviate the namespace path to a single symbol with `:as`
  - Note that [ ] is required when additional options are included
  - This is because we now have a vector of specification elements

```
(ns test.foo
  (:require [test.bar :as bar]))

(bar/myfunc x)  ; we can now use bar instead of test/bar
```

Namespaces…cont's

- We can also eliminate the need to fully qualify specific functions by using the `:refer` option

```
(ns test.foo
  (:require [test.bar :as bar :refer myFunc]))

(myFunc x)          ; myFunc can be called directly
(bar/something)     ; something still needs to be qualified
```

foo.clj, baz.cly, bar.clj

Importing system libraries

- The same logic is used to import libraries from the Clojure framework itself.
- In fact, there are two possibilities
  - Clojure libraries
  - Java libraries/classes
- For Clojure libraries the code is exactly as we have seen already

```
(ns test.foo
  (:require [clojure.string :as str]))

(str/trim "   abc ")  ; => "abc"
```

Importing java classes

- It is also possible to import Java classes into your Clojure code.
- For this, you use `:import` instead of `:require`.
  - Note that you can then use the class without the fully qualified path
- Unfortunately, the syntax to actually use the imported class is a bit ugly
  - `(. (Date.) getTime)`
  - `(Date.)` actually instantiates an object
  - See the code sample below for slightly more readable versions

```
(ns test.foo
  (:import java.util.Date))
```

mods.clj