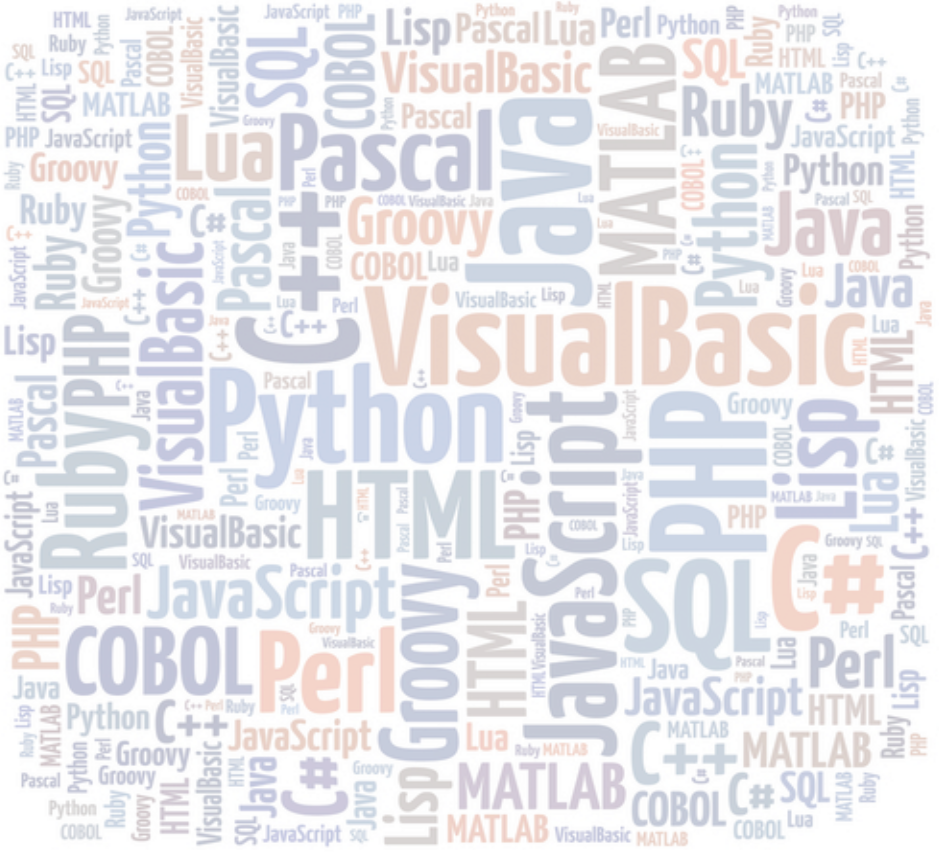
COMP 6411: Comparative Programming Languages

Clojure Functions



Introduction

- Clojure functions
- Basic format
- Parameters
  - Multiple arities
  - Calling other function arities
- Passing and returning functions
- Closures
- Map and reduce
- Core functions
- Lazy sequences

A closer look at functions

- As a functional language, Clojure emphasizes the role of "true" functions.
- So let's look again at the format of a Clojure function

```
; standard form of function
( defn func_name
  "doc string"    ; standard documentation
  [parms]         ; zero or more
  (expression 1) ; function body
  (expression 2)
  …
  (expression n))
```

The details

- Like other languages, Clojure provides mechanisms for documentation
- For internal, programmer-specific documentation, we use the ";"
- For external, user-specific documentation, we typically add doc strings to our functions
  - This is a string located just after the `defn` line
  - Typically, we provide a multi-line string, with the first line being a summary, and remaining lines providing details.

Details…cont'd

- The `doc` function can be used to display documentation for a given method
  - (doc myFunc)
  - Note that in a standalone program, you may have to load the clojure.repl module first
- Note that there are third party tools that can be used to prepare indexed, web-based documentation from the documented source code
- Tools include:
  - Codox
  - Marginalia
  - Cadastre

basic.clj

Parameters

- Clojure functions can take any number of parameters
  - We refer to this as the function's `arity`
- Clojure support *arity overloading* to allow for multi-parameter versions of a function
  - Similar to method overloading in Java, except that types are irrelevant here, only the parm count is relevant

```
; basic arity overloading
( defn func_name
   ([arg1]              ; one parm version
    (expression))

   ([arg1 arg2]         ; two parm version
    (expression)))
```

Arity cont'd

- It is also possible to call other versions of the function, to utilize default functionality
  - Again, similar to what we do in Java with constructors, for example.

```
; using other arities
( defn func_name
   ([arg1]
    (expression))

   ([arg1 arg2]
    ( func_name arg1 ) ; call func with one arg
    (expression) ))
```

multi.clj

Passing functions

- Because functions are first class entities in Clojure, they can be passed as arguments without any special syntax
  - The receiving function simply accepts the function parm like any other value.
  - It is then free to use it inside its function body
  - No special prototypes are needed

```
; passing functions
(defn outer
  [inner data]  ; parm1 = func, parm2 = data
  (inner data)) ; inner() is called on data
```

Anonymous functions

- As previously discussed, anonymous functions can be quite useful in Clojure
- Often we pass a small anonymous function to another function, where it will be used
- The `map` function provides a simple illustration of this
  - In short, map applies its function argument to each of the elements of the associated sequence

```
(map (anonymous function) data)
```

Map corollary

- It is also worth pointing out that functional languages generally provide additional "application" functions.
  - In other words, functions that iteratively process a data structure
- These include
  - Reduce
    - Apply the function parm to all elements of the sequence in order to produce a single value
  - Filter
    - Apply the function parm to all elements individually in order to extract the elements that match the filter

anon.clj

Returning functions

- It is also possible for a function to return another function
  - To do this, the new function definition simply has to be the last expression in the body of the outer function
  - Typically this inner function is anonymous

```
(defn outer
  (expression 1)
  (expression 2)
  (anonymous inner function))
```

Closures

- By itself, this isn't terribly interesting.
- In practice it is often used to create something called *closures*.
  - Partially instantiated functions that store the variables that are currently in the scope of the outer function
  - It is often easiest to understand this with examples

```
(defn outer
  [arg1]
  (fn [] (arg1 * 2)) ;return this func
…
(def c1 (outer 4))    ; arg1 in c1 set to 4
```

Why closures?

- Clojure is not Object Oriented in the way that we think of languages like Java.
  - So there is no concept of a Clojure object that stores instances variables that methods would act upon.
- However, closures provide a similar mechanism in functional programming
- Specifically, they allow one to set “instance variables” in a function invocation

Closures…cont'd

- These versions of the function can then be invoked at a later point.
- When this is done, the function can be used to produce different results in the way that a Object Oriented method can produce a result that is unique to the data in the current instance of the object.
  - For example, a Customer.balance() method would produce different results for each customer since the balance variable would hold a different value for each customer.

clos.clj

Core functions

- We've mentioned core Clojure functions like map, reduce, and filter.
- Let's quickly look at a few more to round out our basic toolkit.
- (`first` *sequence*)
  - Return the data associated with the first element of the sequence (e.g., a vector)
  - Note that there is also a `last` function
  - `(first [4 6 2 1])  ; => 4`
- (`rest` *sequence*)
  - Return a list version of the sequence that includes all elements except the first node
  - `(rest [4 6 2 1])  ; => (6 2 1)`
  - Note that first/rest can be quite useful when writing recursive sequence processing functions

Core…cont'd

- (`cons` item *sequence*)
  - "construct" a new list by prepending the item to the existing sequence
  - `(cons 3 [4 2 5 2])  ; => (3 4 2 5 2)`
- (`take` n *sequence*)
  - Return a list of the first n elements of the sequence
  - `(take 2 [4 3 6 7]) ; => (4 3)`
- (`drop` n `sequence`)
  - Return a list with the first n elements of the sequenced removed
  - `(drop 2 [4 3 6 7])  ; => (6 7)`

Core..cont'd

- (`take-while` func *sequence*)
  - Extract the elements from the sequence until the func result is no longer true (truthy)
  - Here, truthiness is determined by the application of func to each element in the sequence.
  - `(take-while even? [2 4 5 6 7]) ;=> (2 4)`
- As you might expect, there is a `drop-while` as well.
  - `(drop-while even? [2 4 5 6 7]) ;=> (5 6 7)`

Core…cont'd

- A slightly more sophisticated take-while example is given below
- **Note** the following:
  - #( ) is a shorthand for defining anonymous functions
  - The % sign is used to capture the argument passed into the outer function
  - This mechanism can be extended to %1, % 2…% n, etc.

```
(take-while #(< % 5) [ 3 1 4 6 7])
 => (3 1 4)
```

Core…cont'd

- (some func *sequence*)
  - Determines whether func returns a truthy value for at least one element of the sequence
    - In practice, it returns the first truthy value or nil otherwise.
  - (some even? [3 4 5 6 7]) ; => true
  - (some even? [3 5 7]) ; => nil
- (concat seq1 seq2)
  - Appends one sequence to another
  - Returns a final list
  - (concat [1 6 7] `(23 2 4) ; => (1 6 7 23 2 4)

Core…cont'd

- Sorting is quite easy to do as well
  - By default, standard comparison operations will be performed for the relevant type
    - You can't mix types
  - `(sort [4 5 2 7])  ; => (2 4 5 7)`
  - `(sort ["d" "ab" "a"]) ; => ("a" "ab", "d")`
- You can also provide custom comparators
  - The anonymous comparison function below sorts in reverse order

```
(sort #(> %1 %2) [3 1 4 6 7])
 => (7 6 4 3 1)
```

Core…cont'd

- We often want to convert one sequence into another
- It is possible to apply functions like `list` or `vector` to another sequence
  - `(vector '(1 2 3 4))`
  - However, this will treat the list as a single element and this element will be inserted into the vector as one item
  - `=> [(1 2 3 4)]`
- Instead, we should use the `into` function
  - `(into [ ] '(1 2 3 4))  ; => [1 2 3 4]`

Lazy sequences

- In many cases, it would be inefficient to materialize a massive sequence in order to then call a simple function on it.
- In many cases, Clojure will employ "lazy sequences" so that it only materializes what it needs to.
- `first` and `last` provide an illustration of this

```
(first (range 1000000)) ; => 0
; instantaneous response

(last (range 1000000))  ; => 999999
; noticeably slower
```