# COMP 6411: Comparative Programming Languages

## Intro to Clojure

## Introduction

- Clojure and the JVM
- Expressions and forms
- The REPL
- Clojure return values
- Function definitions
  - Anonymous and named
- Control flow
  - If, do and when
- Truthy and Falsey values

## The big picture

- Clojure was essentially designed as a modern dialect of LISP.
  - Its syntax might be described as a cleaned up, modern version of an old language.
- Clojure has several strengths
  - A very simple syntax
    - Initially may seem unintuitive, relative to Java
    - But almost all expressions have the same simple form
  - Implemented on top of the Java Virtual Machine
    - Access to Java classes and components
    - Automatically exploits the JVM's maturity and portability
  - Like functional languages in general, it has direct support for concurrency
    - Well suited for multi-core platforms

## Let’s start with some syntax

- Clojure code is constructed as a series of expressions or *forms*.
- Each expression will consist of two basic elements.
  - Data types or data structures
  - Some operation on this data
- Expressions are always written in parenthesis
  - The first value inside the parenthesis is the operation
  - The remaining values are data that the operation will act upon

## A simple example

- Let’s start by looking at a simple addition
  - (+ 2 2)
- Note the following:
  - The expression is surround by ( )
  - There are no commas (normally) – values are separated by white space
  - Basic math operations use “prefix” style, rather than the more conventional “infix” notation.
- We can treat this as though it were a function invocation
  - The add function takes two arguments.
  - In fact, internally, this is essentially how + is implemented.

## Return values

- All expressions in Clojure return a value
- Often, this is obvious
  - (+ 2 2) clearly should return a 4
- But in other cases, it may not seem obvious what the return value should be.
  - For example, Clojure has syntax for an IF, as well as print functionality.
  - What should these expressions return?
  - For now, just keep in mind that Clojure always returns something, though you don’t necessarily have to use the return value.

## The Clojure REPL

- Before we go any further, it is worth pointing out that Clojure provides a command line interface (like Python)
  - The REPL
    - READ-EVAL-PRINT-LOOP
  - It reads one expression at a time, evaluates it, prints the result and/or return value, and waits for the next expression
  - It is a good way to get used to the basic syntax of the language without having to compile and run code in the IDE

## Nested expressions

- We can nest expressions, as required.
- As noted, each expression returns a value when evaluated.
  - With nested expressions, this essentially means that an inner expression is replaced with its return value and then passed along to the enclosing expression
- How would we write 3 + 2 * 4 – 1
  - ( – (+ 3 (* 2 4) ) 1)
  - 2 * 4 is replaced with 8
  - 3 + 8 is replaced by 11
  - 11 – 1 returns 10

## Some simple examples

```
; simple Math
(+ 1 1)
(- 2 1)
(* 1 2)
(/ 2 1)

; Equality testing
(= 1 1) ; => true
(= 2 1) ; => false

; negation
(not true) ; => false
```

- Note that comments are defined by “;”

## Explicit printing

- When running Clojure code in the REPL, a return value is always printed, following the evaluation of each expression.
- Keep in mind that this is NOT some sort of automatic print statement.
- Clojure provides explicit print forms
  - print *content*
  - println *content*
- Like any expression, print and println must be enclosed in parentheses

## Printing…cont

- A few examples would include

```
(print “woo”)
; => “woo”

(print “woo” “hoo”)
; => “woo hoo”

(print “Number:” 4)
; => “Number: 4”
```

- Note that the print statement itself will return `nil`

## Basic types

- Clojure provides the standard primitive types that you would expect
  - Integer
    - corresponds to long int
  - Float
    - corresponds to double
  - String
    - Always enclosed by double quotes
    - “My big dog”
  - Boolean
    - true or false
  - nil is used as the NULL value
- Types are not explicitly defined
  - Clojure will determine them from context

## Binding names to values

- As previously noted, functional languages do not generally manage global state
- Still, it can be useful to associate values with labels so that they can be referenced later
- In Clojure, this is done as follows:
  - (`def` *name value*)
- Superficially, this is a lot like defining a variable in an imperative language
- However, these are local values (local to the current namespace) that really aren't meant to be changed (though they can be).
  - As we go along, we will see how Clojure handles state information.

## Trivial Functions

- Clojure is a Functional language, so clearly functions are important
- We will talk about functions a lot more very soon, but let’s look at the basic syntax

```
( fn [parm list] body)
```

- So let’s use this to create a *hello world* function

```
( fn [] “hello world”)
```

## Functions…cont'd

- So what is going on here?
  - Note that no function name is provided, so this is actually an *anonymous* function
  - It takes no parameters
  - Its body consists of nothing more than a string
- What does this do?
- Recall that ALL expressions have a return value.
- In Clojure there is no `return` statement.
  - In the case of a function, the return value is always the last expression within the function.
  - So this function returns the string "hello world"
  - You can see this by evaluating it in the interpreter
  - Note that to invoke an anonymous function, you must wrap it in parentheses

## Function names

- Anonymous functions, as we will see later, can be quite useful.
- In many cases, however, we need a function to be named so that we can use it later.
- We can use the `def` keyword to do this with a function

```
( def hello ( fn [] “hello world”) )
```

- Typically, you will use the **defn** mechanism than combines `def` and `fn` into one step

```
( defn hello [] “hello world” )
```

## Control Flow

- Clojure provides basic mechanisms to control the flow of execution.
- This includes:
  - IF
  - DO
  - WHEN
- The logic is conceptually similar to what you see in other languages, though the syntax is a little different

# The IF form

- The basic syntax of an IF is given below:

```
( if boolean-form
     then-form
     optional-else-form)
```

- A practical example might be

```
( if (> 5 0)
    "positive"
    "negative")
; returns "positive"
```

## IF…cont'd

- Again, the else-form is optional, so you can have the following:

```
( if (> 5 0)
    "positive")
; returns "positive"
```

- What about this?

```
( if (< 5 0)
    "negative")
```

- Returns `nil` since the then-form is not executed
  - Remember, all Clojure expression must return a value

## Compound forms

- The previous examples showed a then/else form that executed a single expression.
- What if you need to do more than one thing in the if/else response.
  - You might think that something like the following would work

```
( if (> 5 0)
     “positive”   ; then 1
     “good”       ; then 2
     “negative”)  ; else
```

- But this would be invalid since Clojure can not tell if “good” belongs to the If or the Else

## The Do form

- Clojure provides a Do form in order to a permit a sequence of expressions to be defined.

```
( if (> 5 0)
    (do
        (println “looks good”)
        “positive”)
    (do
        (println “looks bad”)
        “negative”))
```

- Here, each do expression consists of two expressions
  - Of course, any number can be used.

## Do…cont’d

- What is the return value in this case?
- In Clojure, the return value of a compound expression is always the return value of the last expression in the block.
- So the previous example returns the string “positive”
- What if we reverse the two expressions in the first `do`?
  - The return value would be nil since println returns nil.

## Combining IF and DO

- Clojure also provides a when-form that combines IF and DO.

```
( when (> 5 0)
       (println “looks good”)
       “positive”)
```

- The `when` form executes the expression block when the condition is true
  - It always returns nil on false
- Note that there is an implied `do` around the expression block
  - This causes no problem since there is no optional `else` expression

## Truthiness and Falsiness

- As noted, Clojure includes conditional checks.
- Some produce obvious true or false values
  - if (< 5 0)  ; obviously true
- However, because Clojure expressions <u>always</u> return a value, some return values are not true or false in the conventional sense.
- This gives rise to the notion of what Clojure calls *truthy* and *falsey* values.

## True/false…cont'd

- First, recall that `nil` means no value.
- In addition, you can test a value for nil
  - (nil? "boo")  ; => false
  - (nil? nil)    ; => true
- Second, `nil` and `false` are always false (or falsey)
- By extension, everything else is considered true (or truthy).
  - Sometimes this distinction may not always be obvious

## True/false…cont'd

- So what does this expression return?

```
(if (if(println "boo") "woo") "truthy" "falsey")
```

- It  returns "falsey" (and displays boo"falsey"). Why?
- The inner `if` consists of:
  - `(if(println "boo") "woo")`
- The println is always executed as part of the test
  - Recall that println returns nil
  - nil is considered to be a falsey value
  - This means that "woo" will not be printed
  - Moreover, the inner `if` expression will now return nil as the result of the test.
- This means that for the outer `if`, the else-form is executed, thereby returning "falsey".