Week 2: The Clojure Language

Background
Basic structure
A few of the most useful facilities

A modernized Lisp

- Review of Lisp's origins and development
- Why did Lisp need to be modernized?
- Relationship to Java

An insider's opinion

- "In the study of Artificial Intelligence, the reason for learning Lisp is roughly that for learning French if you were going to France it's the native language."
  --Charniak & McDermott, 1985, p. 33

- What do they mean by "native language"?
- Do we believe that?

Their explanation

- "First, Lisp is more flexible than most languages. Users have such total control over what goes on that if they do not like the syntax of the language, they may change it to suit themselves..."

- "The second reason for choosing Lisp is the way in which Lisp is oriented toward the manipulation of symbols as opposed to, say, numbers."

- From today's point of view that's overstated, but it explains how Lisp came to dominate the field.
Homoiconicity

- What's that?

- Lisp code is homoiconic

- Code is data. Programs can:
  - Create and execute other programs on the fly
  - Modify themselves.

- This is a very powerful concept, but also hard to manage.
  - cf. assembly language code

So what's wrong with Lisp?

- Too many dialects.
  - Standardization came late and many "improved" versions of Lisp were being widely used.
  - ANSI Common Lisp (1994)

- Too many parentheses
  - hard to read
  - error prone to code

- Slow execution
  - Interpretive rather than compiled execution

- Original Lisp was naively machine specific
  - Keywords referred to specific properties of the IBM 704 internal instruction format!
  [http://www.idinews.com/history/carcdr.html](http://www.idinews.com/history/carcdr.html)

Clojure objectives and advantages

- Fewer parentheses (but still a lot)
- Faster execution
  - Compiles to Java bytecode.
  - But it remains homoiconic!
  
  Is that possible?

- Full access to Java capabilities

  Do we care about that? Why? When?

- Unlike Lisp, not specifically for A.I, but a general-purpose programming language

  Is that good?

The choice for this course

- We’re going to use Clojure instead of Lisp for our examples because:
  - It avoids some of the ugliness of Lisp (But Clojure is still a little ugly and error-prone)
  - It's available and free.
  - It works on any computer that has a JVM.
  - It generates moderately efficient code

- They're similar enough that if you know Clojure you can pick up Lisp very quickly if you ever need it.
Review

- We've already seen (session 1) that:
  - Code and data are both written as lists
  - A function invocation is a list in which the first element names the function and the rest of the elements are the arguments: Example:
    `(power 2 24)`
  - Commas are allowed but not needed.
  - Simple (i.e. short, one-line) functions can be invoked (and defined) from the REPL command line.

**So how do we create and use more interesting functions?**

Getting started

- This little web-based tutorial
  is a very simple introduction to command-line (REPL) dialogues

- It prompts you to follow specific instructions, but you can also experiment a bit on your own.

- It may help you to get used to the behavior of Clojure and the strange syntax.

- Note the error messages (from Java!) when you enter something illegal.
  - Some of them are clear.
  - Others aren't but you'll get used to them.

Functions (basics, but ugly)

- Running a Lisp or Clojure program consists of executing a defined top-level function, which may, of course, invoke other functions.

- To define a function `(fn [args] returned-value)` but it has no name, an anonymous function!
  - You can define `(fn [x] (* x x))` but it doesn't do anything useful. You can say `((fn [x] (* x x)) 8)` and get the result 64, but that's tedious, inelegant, and error-prone.

**When might you ever define and use an anonymous function?**

Functions (named)

- You could combine the facility for defining a symbol and the definition of an anonymous function:
  `(def square (fn [x] (* x x )))`
  and then invoke it `(square 8)` but there's an easier and clearer way.

- `(defn square [x] (* x x ))`
  or more generally
  `(defn fctnname [arglist] value)`

**Note:** `defn` is a macro (but do we care?)
A trivial interaction with REPL
(from the tryclj tutorial)

> (defn square [x] (* x x))
#'sandbox34619/square
> (square 0.5)
0.25
> (square 9)
81

Looks fine!
What could possibly go wrong?

An irritating setback

- After experimenting with other Clojure features, we try to repeat the earlier use of square

  > (square 9)
  java.lang.IllegalStateException: Attempting to call unbound fn: #'sandbox30082/square

- We somehow lost our working directory! None of the functions we've defined can be found!
  - There's very likely a simple answer to what happened and an easy way of recovering, but we may have to wait until next week to discover exactly what to do.
  - Fortunately you don't have to use Clojure this week.

But we can still continue exploring Clojure

- We have to shut down the "try Clojure" web page and restart it.
  - That's irritating, but later we'll learn how to avoid whatever we did.
  - (I suspect a bug in the REPL)
  - Fortunately, everything is working again.
  - Other tools have similar irritating peculiarities, but we've gotten used to them.

Another simple function

- (defn average [numbers]
  (/ (apply + numbers)
      (count numbers)))
#'sandbox7982/average

  > (average [60 80 100 400])
  160

What's different about the above from the usual REPL interaction?
Let's explore more Clojure

- Let's focus on **conditionals**, because they play a central role in inference engines and other AI applications.

- But Clojure and Lisp experts warn us to think of conditional evaluation (if, cond, when, etc.) not as controlling **sequence of execution** but just as determining **precedence of definition**.
  - Don't worry too much about that.
  - In many situations (esp. with immutable functions) it amounts to the same thing.

Testing truth value

- As in LISP, the principal and most useful conditional facility is **cond**
  - It uses fewer parentheses than the LISP version, but it's just as powerful and flexible.

- Yes, it's a **function**, not an imperative operator or command verb.
  - Actually it's a **macro** on top of **if**
  - Easier to organize and read complicated code than using if directly.

- We'll need it if we want to use Clojure for assignment #1 (but that wasn't required)

if tests a single condition

- Two or three operands:
  - `(if condition trueResult falseResult)`
  - `(if condition trueResult)`

- The condition is not necessarily a pure Boolean expression. Anything other than **nil** or **false** is considered true!
  - Is that a help or a confusion?

- If no **falseResult** is specified and **condition** is **false**, the result is **nil**.

cond tests multiple conditions

- " Takes a set of test/expr pairs. It evaluates each test one at a time. If a test returns logical true, cond evaluates and returns the value of the corresponding expr and doesn't evaluate any of the other tests or exprs. (cond) returns nil."
  - from the Clojure.docs web site

- Here's a simple example:
  - `(cond (age < 18) juvenile
            (age < 23) student
            (age < 65) adult
            :else senior)`

- Note: Unlike switch case in other languages the clauses needn't test values of the same variable.
- Often used effectively with recursive function invocation (more next week)
Improving on Lisp

- The Clojure conditional
  \[
  \text{(cond} \begin{align*}
  & (\text{age} < 18) \quad \text{juvenile} \\
  & (\text{age} < 23) \quad \text{student} \\
  & (\text{age} < 65) \quad \text{adult} \\
  & \quad \text{: else} \quad \text{senior})
  \end{align*}
  \]

  is simpler and less error-prone than the Lisp version:
  \[
  \text{(cond} \begin{align*}
  & ((\text{age} < 18) \quad \text{juvenile}) \\
  & ((\text{age} < 23) \quad \text{student}) \\
  & ((\text{age} < 65) \quad \text{adult}) \\
  & (T \quad \text{senior}))
  \end{align*}
  \]

Another syntax improvement

- When Clojure or Lisp sees a list, it wants to evaluate it:
  - The first element names a function
  - The remaining elements are the arguments

- But sometimes a list is just data.
  - How can we suppress evaluation?

Suppressing list evaluation

- In early Lisp you'd say `quote (list)`
  - That still works in Clojure, but that's ugly and tedious to code. It originated when character sets were limited.
  - Today we prefer to say the more readable `' (list)`

- Also Clojure supports vectors `[1 2 3 4]`
  - for example as function parameter lists and some special forms that don't evaluate the first element as a function call.

Summary

- We've shown a few of the most common and most useful features of Clojure.

- From here on you should be able to read and understand the Matuszek handbook (web link from the course schedule page)
  - Let's look it over now

- Starting next week, we'll concentrate on A.I. concepts and coding.
  - See schedule page (still under development but getting fleshed out.)