Part 2, Week 8: The Clojure we need

- In order:
  - to do the exercises
  - to understand the examples and principles of search programs used in A.I.

- But avoiding:
  - advanced features that aren't essential for introductory A.I. applications
  - Java interfaces

- First, a little background and motivation . . .

Programming languages large and small

- A large programming language is designed to provide all the tools a programmer needs to solve a wide range of problems (applications).
- Examples -- In that sense these languages were designed large:
  - PL/I
  - Ada
  - ALGOL-68

- What happened to them? Why aren't we using them more today?

- So, what's a small programming language?

Programming languages large and small

- A small programming language is designed:
  - to solve a limited class of problem
  - to be mastered quickly (in an afternoon!)
  - with few surprises (no arcane rules)

- Some are intended to be complete. Examples:
  - original (Kemeny) BASIC
  - original Java
  - early FORTRAN

- Others were designed to be extensible. Example:
  - LISP

An unexpected (but surprisingly common) problem

- Many small programming languages gradually expand and become large!
  - New features are added
  - Additional paradigms are supported
  - People use it for problems it wasn't designed for

- Examples:
  - C has spawned C++, Java, and C#.
  - BASIC has spawned VisualBasic

- Each of those, with its essential libraries, is now much bigger and more complicated than any of the languages that used to get condemned for being too big ("bloated")!
In particular LISP

- Began as a very small language, ca. 1958
  - The original idea was that a simple, expandable, and logical language could be applied to support extremely sophisticated processes.
- But then features were added to various versions of LISP almost continuously. Of particular interest were:
  - various macros
  - CLOS (common LISP object system) for OOP
- ANSI Common LISP was standardized in 1994! Too late! LISP had become huge!

What about Clojure?

- Clojure wanted (needed) to compete with (replace) LISP.
- But avoiding LISP's mistakes. Clojure had to be easier than LISP:
  - for programmers to learn quickly
  - to implement cheaply and reliably
- Being built on top of Java helped to avoid some implementation complexities. Clojure can take advantage of:
  - JVM capabilities
  - Java libraries, built-in data types, exception handling, etc.

But unfortunately

- Clojure is still a rather large language
- There isn't yet (or we just haven't found) a suitable textbook for Clojure programming:
  - Halloway is sloppy and full of minor errors
  - Emerick, is confusingly sequenced & badly written
  - Neither is specifically oriented to A.I.!
  - I just ordered this one http://www.manning.com/fogus/Sample-Ch9.pdf
- Do we regret choosing Clojure for our course? Not at all! Why?
- Let's look over the whole language, emphasizing what we're most likely to need now.

Comments

- As in every programming language, clear comments are essential to an understandable program.
- The semicolon is the usual comment delimiter
  - It can introduce a whole comment line or a comment to the right of any code.
  - We saw examples in the power function sample solution.
  - It's like // in the C family
- There's are two other forms of commentary, which we won't need now.
Commentary example

(defn power [x n]                          ; Integer power function
  cond (= 0 n) 1                          ; base
  (= 1 n) x                                ; cases
  (Remainder not shown, pending assignment #1)
)

Notes:

- Instead of the comment on the first line, we could code a longer, more descriptive comment on full lines before the code
  ; Integer power function
  ; This function raises a real number to an integer power
  (defn power [x n])
- We emphasize the purpose of an expression, not how it works
  (- n 1) ; Subtract 1 from the current n
  No! The reader should know that.

Documentation expectation

- In this course*, a program that produces the correct result will earn at least a C grade.
- An A program, is clear and understandable to anyone who knows the programming language.
  - But don't explain what's obvious!
  - Focus on what and why more than how.
- * Of course understandability should apply beyond this course to every program we create in any language.

Macros

- Some programming languages support macros.
- A macro is a symbol (possibly with parameters) that is replaced by other (usually longer and more complicated) source code before the actual compilation.

Macros in various languages

- Languages that support macros:
  - Many large assembly languages
  - PL/I
  - C and C++ (crudely)
  - Lisp
  - Clojure
- In Clojure invoking a macro looks like calling a function, but instead of compiling to Java byte code it expands to other lower-level legal Clojure source code.
  - The expanded code is usually longer and more complicated than the macro call. (Otherwise why bother with a macro?)
Macros versus functions

- "Macros operate at compile-time. This means that they are not first-class citizens of a running Clojure program like functions are. A macro has no access to runtime information, such as the current values of a var. [But the code the macro generates may.] A macro sees only unevaluated data structures read from source code." -- Emerick, Carper, & Grand p. 243

- The good news: We won't necessarily care about that distinction for our simple exercises (but be aware of it, esp. if you get an error message that isn't clear).

So, we can think of Clojure as a 4-level tool

- Macros are defined in terms of functions (and other macros)
- Functions are defined in terms of special forms (and other functions)
- Special forms (just the absolute minimum to make it work) are defined in terms of JVM
- JVM is defined in terms of specific instructions for a given type of computer system.

That's a bit oversimplified, but it helps us in understanding the relationships.

Special forms

- A Clojure form
  
  \[(fctn p1 p2 . . . )\]
  
  is either
  
  - Invocation of a function named fctn with parameters p1, etc., or
  - a special form named fctn

- Special forms are the lowest-level primitive Clojure constructs.
  
  - They can't be defined in terms of other Clojure code.
  
  See http://clojure.org/special_forms

defn

- Although we think of defn as a rather basic primitive operation, it's really a macro which expands into two special forms. an fn inside a def.

- That's good, of course, because it's much more natural and convenient to use.
Immutable functions

- Clojure strongly encourages writing functions that have no mutable state, i.e. no variables whose values change.
  - That simplifies debugging and understandability, but may impose inconvenient restrictions.
  - The easy availability of recursion compensates for that limitation.

- Background:
  - Some experts (Joshua Bloch on Java) favor immutable functions in procedural languages, too.
  - Others (Bjarne Stroustrup) advise making classes behave like primitive code.

  Can they both be right?

Building a database

defstruct

- defstruct is the simplest of several tools for defining a fixed collection (record)

- Example from a music catalogue:
  (extremely oversimplified for assignment #2)

  (defstruct Work :composer :title :duration :performer)

  (struct :composer "Beethoven, L" :title "Sonata #23" :duration 25 :performer piano)

  Does that make sense? What’s wrong with it?

Example (continued)

- (defstruct Recording :title work :performer artist)
  (Recording :title "Sonata #23" :performer "Emanuel Ax")

  Does that make sense? What’s wrong with it?

Two ways of specifying parameters to a function or a macro

- **Keyword** parameters (as in the preceding example) are appropriate when:
  - There are too many parameters to expect user-programmers to remember their sequence.
  - Some parameters are optional (may be omitted)

- **Positional** parameters (as in the power example) are appropriate for short (1, 2, or 3) parameter lists in a natural order that user-programmers can easily remember.
  - They can sometimes be **mixed** (how?)

  The C-family has been criticized for not supporting keyword function parameters.
  - e.g, 12 parameters to a constructor call!
Further note on assignment #2

- The assignment specification makes clear that your code doesn't have to work.

- We just want to get used to some of the concepts of defining Clojure code for:
  - a database
  - predicates on that database

- In assignment #3 we'll go farther.
  - You'll get feedback on assignment #2 before assignment #3 is due.
  - We'll look at more concrete examples, including a sample solution to assignment #2 next week.