Session 7: Review & further explanations

A summary of the functional paradigm and Clojure techniques

COMP 378 -- Spring, 2016
Mr. Weisert

The functional paradigm

- A program is a collection of functions, some of which invoke other functions (i.e., a hierarchy).
- Execution consists of evaluating the top-level function or expression.
- Sounds too simple! What's the catch?

Forward chaining concept

- A rule of the form $P \rightarrow Q$ is said to fire when it's evaluated and $P$ is true.
  - That asserts that $Q$ is also true, which may then cause additional rules to fire.
- When any rule function is evaluated, it may cause other functions to fire, either:
  a. by invoking them directly,
  b. by causing certain conditions to become true
- Clojure supports a variety of ways of expressing $P \rightarrow Q$:
  - if, when, and (most generally useful) cond

Let's explore more Clojure

- Let's focus on conditionals, because they play a central role in inference engines and other A.I. 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.
The **cond** macro
- May specify an unlimited number of conditions and corresponding values.
  - The result of evaluating a **cond** is the value corresponding to the *first true* condition.
- General form
  - `(cond  boolean-expression1  value1
    boolean-expression2  value2
    ...
    boolean-expressionn  valuen)

**Style (format) recommendations for cond**
- Show structure with indentation and alignment.
- For example, whenever possible:
  - Don't put multiple condition-value pairs on the same line
  - Balance parentheses on each line
  - Vertically align corresponding or similar elements
  - Vertically align matching parentheses or bracket pairs that have to span multiple lines.

Are those good rules for any free-form programming language?

**Unfortunately editors and compilers don’t support multiple colors**
- \[ x^{24} = (((x(x(x)))^2)^2)^2 \]
- But many of them will highlight or flash a matching parenthesis or bracket when we put the cursor on one.
  - That helps the programmer to confirm balance or to diagnose unintended imbalance.
  - Clojure’s heavy reliance on parentheses (LISP is worse) may appear intimidating at first, but programmers like it after they get used to it.

**Recursion with cond**
- a common combination
- Some of the value clauses can invoke the current function recursively!
  - Any **base cases** that don't usually come first.
  - Recursive invocations should be for a *much* simpler (smaller) case than the current level
    - *(not just 1 less! Why is factorial an inappropriate function to code recursively?)*
- Recursion is an extremely powerful technique.
  - Use it when it simplifies or clarifies a program.
  - But remember that it carries heavy overhead.
Important distinction

- **In imperative** (procedural) programming:
  - conditions govern *actions*:
    - if *something is true*
    - then *do something*  
      (i.e. cause something to happen)

- **In functional** programming:
  - conditions govern *assertions*:
    - if *something is true*
    - then *assert a fact*  
      (i.e. assume something else is true)

- **But** some functions can have side effects
  - Discouraged in functional programming but occasionally useful, even necessary.

---

Functions in LISP
(Clojure, too)

- A function can create and invoke other functions!
  - That can be extremely powerful.

- Recursion is assumed
- Multi-threading is assumed

---

Immutable functions

- The functional paradigm strongly prefers functions that don't modify the initial values of their internal state, because:
  - Multithreading can't corrupt them. Programmers don't have to worry about synchronization, protection, etc.
  - Immutable functions are often easier to understand and therefore less likely to have subtle bugs.

- **But:**
  - Simulating iteration with recursion can be extremely inefficient.
  - Some obvious programming techniques are then unavailable.

---

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?*
In particular

- Certain convenient and efficient program constructs conflict with immutability:
  - Compound assignment operators in the C-family of languages (+=, -=, *=, /=, %=, |=, &=, ++)
  - While / for / do iteration that depends on an updated condition for termination.
- But those are powerful and useful facilities for procedural and object-oriented programming.
- Advice (warning: controversial)
  - Favor immutability in inference-engine and similar A.I. applications as well as with high-performance multi-tasking.
  - Forget about it in traditional single-thread logic!

Tail recursion

- When a recursive function call is the last executable statement in a function, the compiler can better optimize the code, since it needn't restore the local environment after the recursive invocation returns.
- A recursive invocation inside a return statement is a typical way of doing tail recursion.
- We saw that the ? : construct in the C-family supports conditional tail recursion.

Homoiconicity

- What's that?
- Lisp code is homoiconic
- Code is data. Programs can:
  - Create and execute other programs on the fly
  - Modify themselves.
- Is that good or bad?

Self-modifying programs

- A powerful concept, but hard to manage.
  - cf. old-fashioned assembly (machine) language code
  - Self modifying programs were once considered extremely sophisticated; something to boast about!
  - But they're extremely difficult, sometimes impossible, to understand!
- Structured revolution (~1978) put an end to that in traditional procedural languages:
- A few remnants survived:
  - COBOL's ALTER statement modifying the target of a GO TO statement.
  - FORTRAN's similar ASSIGNED GO TO.

But . . .
The LISP world wasn't convinced
- Drawing upon the homoiconicity concept, A.I. specialists continued to exploit self-modifying and dynamically-generated code.
- The Clojure world has inherited those practices.
- Exploit them, but in a disciplined and understandable way.

Basic forward-chaining search problems
- Given:
  - a set of facts (data base)
  - a set of rules (knowledge base)
  - a desired or possible conclusion*

- Determine whether the conclusion is:
  - implied by (a consequence of) the facts,
  - inconsistent with the facts (contradictory),
  - neither

* If no conclusion is specified, determine whatever logically follows from the facts.

Representing facts and forward chaining inference
- Whenever the inference engine evaluates a rule with true condition, we say that the rule fires.
- That may then make it possible for other rules in the data base to fire, a chain reaction.
  - Obviously, this is interesting for A.I. only with huge numbers of predicates.

Clojure conventions for logical constants
- In Clojure anything that isn't false or nil is considered to be true.
- Note that even 0 is true!
- Predicates yield either true or false as expected.
  - Relations (=, <, etc.)
  - Forms that yield true or false (conventionally using suffix question mark in the name).
Pattern matching

- In general, a pattern matches some expression (a possibly nested list) if there is a binding of variable names to values such that substituting those values into the pattern yields the expression. --John DeNero

- For example the expression \(((a \ b) \ c \ (a \ b))\)
  - matches the pattern \( (?x \ c \ ?x)\)
    - with variable ?x bound to value (a b).
  - and also matches the pattern
    \( ((a \ ?y) \ ?z \ (a \ b))\)
    - with variable ?y bound to b and ?z bound to c.

Do we agree and understand?

What we need to know about unifiers

- What they are, what they do.
- Why they're useful.
- How to use one.

- But we're not necessarily going to build our own.
  - Unless we find a helpful and clearly understandable Clojure model / example

Simplified unification algorithm
(from composingprograms web site)

1. Both inputs e and f are replaced by their values if they are variables.
2. If e and f are equal, unification succeeds.
3. If e is a variable, unification succeeds and e is bound to f.
4. If f is a variable, unification succeeds and f is bound to e.
5. If neither is a variable, both are not lists, and they are not equal, then e and f cannot be unified, and so unification fails.
6. If none of these cases holds, then e and f are both pairs [i.e. lists--why?], and so unification is performed on both their first and second corresponding elements.

Rather clear except for item 6

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 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).

Special forms

- A Clojure form
  
  \( (fctn \ p1 \ p2 \ldots) \)

  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

Input-Output and user-interface code

- Don't worry about it in this course.

- Just focus on the logic, inference engine, properties, etc.

- When you develop a real application, you can incorporate a sophisticated user interface, perhaps based upon Java facilities.

Questions?

Comments?

Complaints?