INF4820: Algorithms for Artificial Intelligence and Natural Language Processing

Common Lisp Core

Stephan Oepen & Murhaf Fares

Language Technology Group (LTG)

September 8, 2016

Agenda

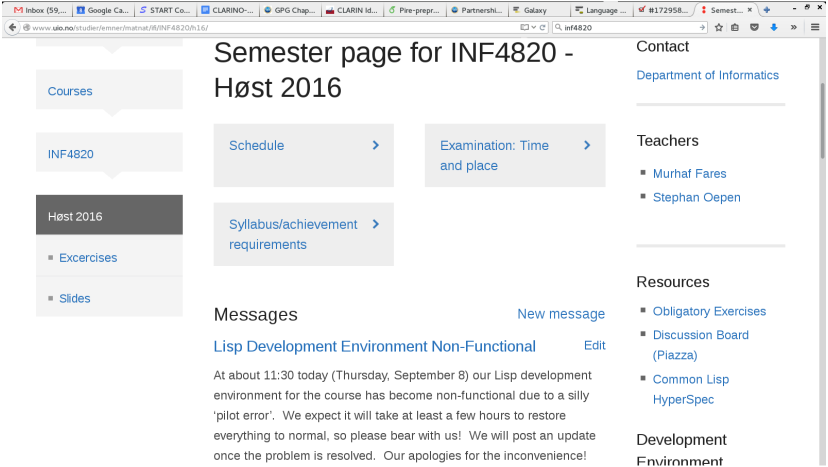
Previously

- Common Lisp essentials
- S-expressions (= atoms or lists of s-expressions)
- Recursion
- Quote
- List processing
- Identity vs. Equality

Today

- More Common Lisp
- Higher-order functions
- Argument lists
- Iteration: (the mighty) `loop`
- Additional data structures

Did You Check the Course Page Today?



http://www.uio.no/studier/emner/matnat/ifi/INF4820/h16/

Rewind: A Note on Symbol Semantics

- Symbols can have values as functions and variables at the same time.
- #' (*sharp-quote*) gives us the function object bound to a symbol.

```
? (defun foo (x)
    (* x 1000))

? (defparameter foo 42) → 2

? (foo foo) → 42000

? foo → 42

? #'foo → #<Interpreted Function FOO>

? (funcall #'foo foo) → 42000
```

- #' and `funcall` (as well as `apply`) are useful when passing around functions as arguments.

Higher-Order Functions

- Functions that accept functions as arguments or return values.
- Functions in Lisp are first-class objects.
  - Can be created at run-time, passed as arguments, returned as values, stored in variables . . . just like any other type of data.

```
? (defun filter (list test)
    (cond ((null list) nil)
          ((funcall test (first list))
           (cons (first list) (filter (rest list) test)))
          (t (filter (rest list) test))))

? (defparameter foo '(11 22 33 44 55))

? (filter foo #'evenp)
→ (22 44)
```

- Functions, recursion, conditionals, predicates, lists for code and data.

Anonymous Functions

- We can also pass function arguments without first binding them to a name, using lambda expressions: `(lambda (`*parameters*`)` *body*`)`
- A function definition without the `defun` and *symbol* part.

```
? (filter foo
          #'(lambda (x)
              (and (> x 20)
                   (< x 50))))
→ (22 33 44)
```

- Typically used for ad-hoc functions that are only locally relevant and simple enough to be expressed inline.
- Or, when constructing functions as return values.

Returning Functions

- We have seen how to create anonymous functions using `lambda` and pass them as arguments.
- So we can combine that with a function that itself returns another function (which we then bind to a variable).

```
? (defparameter foo '(11 22 33 44 55))

? (defun make-range-test (lower upper)
    #'(lambda (x)
        (and (> x lower)
             (< x upper))))


? (filter foo (make-range-test 10 30))

→ (11 22)
```

Parameter Lists: Variable Arities and Naming

Optional Parameters

```
? (defun foo (x &optional y (z 42))
    (list x y z))

? (foo 1) → (1 nil 42)

? (foo 1 2 3) → (1 2 3)
```

Keyword Parameters

```
? (defun foo (x &key y (z 42))
    (list x y z))

? (foo 1) → (1 nil 42)

? (foo 1 :z 3 :y 2) → (1 2 3)
```

Rest Parameters

```
? (defun avg (x &rest rest)
    (let ((numbers (cons x rest)))
      (/ (apply #'+ numbers)
         (length numbers))))

? (avg 3) → 3

? (avg 1 2 3 4 5 6 7) → 4
```

Recap: Equality for One and All

- eq tests object identity; it is not useful for numbers or characters.
- eql is like eq, but well-defined on numbers and characters.
- equal tests structural equivalence
- equalp is like equal but insensitive to case and numeric type.

```
? (eq (list 1 2 3) '(1 2 3)) → nil
? (equal (list 1 2 3) '(1 2 3)) → t
? (eq 42 42) → ? [implementation-dependent]
? (eql 42 42) → t
? (eql 42 42.0) → nil
? (equalp 42 42.0) → t
? (equal "foo" "foo") → t
? (equalp "FOO" "foo") → t
```

- Also many type-specialized tests like =, string=, etc.

From the 2013 Final Exam

Write two versions of a function `swap`; one based on recursion and one based on iteration. The function should take three parameters—x, y and `list`— where the goal is to replace every element matching x with y in the list `list`. Here is an example of the expected behavior:

```
? (swap "foo" "bar" '("zap" "foo" "foo" "zap" "foo"))
→ ("zap" "bar" "bar" "zap" "bar")
```

Try to avoid using destructive operations if you can. [7 points]

A Brief Detour: Macros

- Elevator Pitch: programs that generate programs.
- Macros provide a way for our code to manipulate itself (before it is passed to the compiler).
- Can implement transformations that extend the syntax of the language.
- Allows us to control (or even prevent) the evaluation of arguments.
- We have already encountered some built-in Common Lisp macros: `and`, `or`, `if`, `cond`, `defun`, `setf`, etc.
- Although macro writing is out of the scope of this course, we will look at perhaps the best example of how macros can redefine the syntax of the language—for good or for worse, depending on who you ask:
  - `loop`

Iteration

- While recursion is a powerful control structure,
- sometimes *iteration* comes more natural.
- dolist and dotimes are fine for simple iteration.
- But (the mighty) loop is much more general and versatile.

```
(let ((result nil))
  (dolist (x '(0 1 2 3 4 5))
    (when (evenp x)
      (push x result)))
  (reverse result))
```

→ *(0 2 4)*

```
(let ((result nil))
  (dotimes (x 6)
    (when (evenp x)
      (push x result)))
  (reverse result))
```

→ *(0 2 4)*

```
(loop
    for x below 6
    when (evenp x)
    collect x)
```

→ *(0 2 4)*

Iteration with `loop`

```
(loop
    for i from 10 to 50 by 10
    collect i)

→ (10 20 30 40 50)
```

- Illustrates the power of syntax extension through macros;
- `loop` is basically a mini-language for iteration.
- Reduced uniformity: different syntax based on special keywords.
- Paul Graham on `loop`: “one of the worst flaws in Common Lisp”.
- But non-Lispy as it may be, `loop` is extremely general and powerful!

loop: A Few More Examples

```
? (loop
      for i below 10
      when (oddp i)
      sum i)

→ 25
```

```
? (loop for x across "foo" collect x)

→ (#\f #\o #\o)
```

```
? (loop
      with foo = '(a b c d)
      for i in foo
      for j from 0
      until (eq i 'c)
      do (format t "~a: ~a ~%" j i))
⤳
0: A
1: B
```

loop: Even More Examples

```
? (loop
      for i below 10
      if (evenp i)
      collect i into evens
      else
      collect i into odds
      finally (return (list evens odds)))

→ ((0 2 4 6 8) (1 3 5 7 9))
```

```
? (loop
      for value being each hash-value of *dictionary*
      using (hash-key key)
      do (format t "~&~a -> ~a" key value))
```

`loop`: The Swiss Army Knife of Iteration

- Iteration over lists or vectors: `for` *symbol* { `in` | `on` | `across` } *list*
- Counting through ranges:
  `for` *symbol* [`from` *number*] { `to` | `downto` } *number* [`by` *number*]
- Iteration over hash tables:
  `for` *symbol* `being each` { `hash-key` | `hash-value` } `in` *hash table*
- Stepwise computation: `for` *symbol* `=` *sexp* `then` *sexp*
- Accumulation: { `collect` | `append` | `sum` | `minimize` | `count` | ... } *sexp*
- Control: { `while` | `until` | `repeat` | `when` | `unless` | ... } *sexp*
- Local variables: `with` *symbol* `=` *sexp*
- Initialization and finalization: { `initially` | `finally` } $sexp^{+}$

- All of these can be combined freely, e.g. iterating through a list, counting a range, and stepwise computation, all in parallel.
- Note: without at least one accumulator, `loop` will only return `nil`.