

*INF4820: Algorithms for  
Artificial Intelligence and  
Natural Language Processing*

The Common Lisp Core

Stephan Oepen & Milen Kouylekov

Language Technology Group (LTG)

September 3, 2014





## 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`

# Did You Check the Course Page Today?



The screenshot shows a web browser window displaying the University of Oslo website. The browser's address bar shows the URL [www.uio.no/studier/emner/matnat/ifi/INF4820/h14/](http://www.uio.no/studier/emner/matnat/ifi/INF4820/h14/). The website header includes the University of Oslo logo and name, a search bar, and navigation links for Home, Research, Studies, Student Life, Services and tools, About UiO, and People. The main content area is titled "Semester page for INF4820 - Høst 2014" and is organized into several sections:

- Left sidebar:** A vertical menu with links for "Studies", "Courses", "INF4820", and "Høst 2014". Under "Høst 2014", there are sub-links for "exercises" and "slides".
- Main content area:**
  - Schedule** and **Examination: Time and place** (links)
  - Syllabus/achievement requirements** (link)
  - Requirements**
    - Obligatory Assignments
    - General Regulations
    - Sample Exam
  - Additional Resources**
    - Common Lisp the Language
    - Common Lisp HyperSpec
    - Discussion & Self-Help Group
  - Lisp in the 21st Century**
    - Paul Graham: Beating the Average
    - Paul Graham: Revenge of the Nerds
    - Carl de Marcken: Inside Orbitz
- Right sidebar:**
  - Contact**: Department of Informatics
  - Teachers**:
    - Stephan Oepen
    - Milen Koulyekov
  - Messages**:
    - Self-Help Group (Sep 2, 2014 09:24 PM)
    - First Assignment (Aug 29, 2014 04:34 PM)
    - Screen Casts (Aug 21, 2014 06:50 PM)
    - Lecture Materials (Aug 20, 2014 10:38 PM)
    - Course Schedule (Aug 12, 2014 10:52 AM)

<http://www.uio.no/studier/emner/matnat/ifi/INF4820/h14/>

# Another Communication Channel: Student Self-Help



The screenshot shows a Facebook group page for 'INF4820: Algorithms for AI and NLP (University of Oslo)'. The page features a blue header with the group name and a search bar. On the left, there is a navigation menu with options like 'News Feed', 'Messages', 'Events', and 'Find Friends'. The main content area displays a comic strip with panels containing text such as 'LAST NIGHT I DROPPED OFF WHILE READING A LISP BOOK.' and 'AT LAST, JUST LIKE THEY SAID, I FELT A GREAT ENLIGHTENMENT. I SAW THE NAKED STRUCTURE OF LISP CODE. UNFOLD BEFORE ME.' Below the comic, there are buttons for 'Joined', 'Share', and 'Notifications'. The page also shows a list of members, including Stephan Oepen, Milen Kouylekov, Erik Velldal, Petter Hohle, Per Wessel, Ailsa Aspås, Bendik Segrov Ibenholt, Mathias Ramm, and Jarle Fosén. On the right, there is an 'ABOUT' section with 16 members and a 'CREATE NEW GROUPS' section.

<https://www.facebook.com/groups/287615961438426/>



- ▶ 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.**



- ▶ 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.



- ▶ 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)
```



## 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.
- ▶ `eq1` 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]
```

```
? (eq1 42 42) → t
```

```
? (eq1 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]*



- ▶ Pitch: **programs that generate programs**.
- ▶ Macros provide a way for our code to manipulate itself (before it's 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**



- ▶ 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)
```



```
(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