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

## Common Lisp Data Structures

Stephan Oepen & Milen Kouylekov

Language Technology Group (LTG)

September 4, 2014

# Agenda

## Previously

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

## Today

- More of (the mighty) `loop`
- Input and output via streams
- Data structures: plists, alists, hash tables, structures
- A Common Lisp quiz
- Some suggestions for best coding practises

# `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`.

# loop: A Few More Examples

```
? (loop for foo in '(1 2 3) collect foo)
→ (1 2 3))
```

```
? (loop for foo on '(1 2 3) collect foo)
→ ((1 2 3) (2 3) (3))
```

```
? (loop for foo on '(1 2 3) append foo)
→ (1 2 3 2 3 3)
```

```
? (loop
      for i from 2 to 10
      when (evenp i)
      collect i into evens
      else collect i into odds
      finally (return (list evens odds)))
→ ((2 4 6 8 10) (1 3 5 7 9))
```

# Input and Output

- Reading and writing is mediated through *streams*.
- The symbol `t` indicates the default stream, the terminal.

```
? (format t "~a is the ~a.~%" 42 "answer")
↝ 42 is the answer.
→ nil
```

- `(read-line` *`stream`* `nil)` reads one line of text from *stream*, returning it as a string.
- `(read` *`stream`* `nil)` reads one well-formed s-expression.
- The second reader argument asks to return `nil` upon end-of-file.

```
(with-open-file (stream "sample.txt" :direction :input)
  (loop
      for line = (read-line stream nil)
      while line do (format t "~a~%" line)))
```

# More Data Structures: Arrays

- Integer-indexed container (indices count from zero)

```
? (defparameter array (make-array 5)) → #(nil nil nil nil nil)
? (setf (aref array 0) 42)→ 42
? array → #(42 nil nil nil nil)
```

- Can be fixed-sized (default) or dynamically adjustable.
- Can also represent rectangular 'grids' of multiple dimensions:

```
? (defparameter array (make-array '(2 5) :initial-element 0))
  → #((0 0 0 0 0) (0 0 0 0 0))

? (incf (aref array 1 2)) → 1
```

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 |

# Arrays: Specializations and Generalizations

- *Vectors* = specialized type of arrays: one-dimensional.
- *Strings* = specialized type of vectors (similarly: bit vectors).
- Vectors and lists are subtypes of an abstract data type *sequence*.
- Large number of built-in *sequence functions*, e.g.:

```
? (length "foo") → 3
? (elt "foo" 0) → #\f
? (count-if #'numberp '(1 a "2" 3 (b))) → 2
? (subseq "foobar" 3 6) → "bar"
? (substitute #\a #\o "hoho") → "haha"
? (remove 'a '(a b b a)) → (b b)
? (some #'listp '(1 a "2" 3 (b))) → t
? (sort '(1 2 1 3 1 0) #'<) → (0 1 1 1 2 3)
```

- Others: `position`, `every`, `count`, `remove-if`, `find`, `merge`, `map`, `reverse`, `concatenate`, `reduce`, ...

# Sequences, Function Objects, Keyword Parameters

```
(member "foo" '("foo" "baz" "bar" "c" "a" "b" "xy" "yz"))
→ nil

(member "foo" '("foo" "baz" "bar" "c" "a" "b" "xy" "yz") :test #'equal)
→ nil

(defparameter foo '("foo" "baz" "bar" "c" "a" "b" "xy" "yz"))
(sort foo #'(lambda (x y)
              (let ((i (length x)) (j (length y)))
                (or (< i j) (and (= i j) (string< x y))))))
→ ("a" "b" "c" "xy" "yz" "bar" "baz" "foo")

(defparameter bar '(("baz" 23) ("bar" 47) ("foo" 11)))
(sort bar #'< :key #'(lambda (foo) (first (rest foo))))
→ (("foo" 11) ("baz" 23) ("bar" 47))
```

- Parameterization through higher-order functions as keyword parameters.
- When meaningful, built-in functions allow `:test`, `:key`, `:start`, etc.
- Use function objects of built-in, user-defined, or anonymous functions.

# Associative Key–Value Look-Up

- Several built-in possibilities.
- In order of increasing power:
  - Plists (property lists)
  - Alists (association lists)
  - Hash Tables

# Plists (Property Lists)

- A property list is a list of alternating keys and values:

```
? (defparameter plist (list :artist "Elvis" :title "Blue Hawaii"))
? (getf plist :artist) → "Elvis"
? (getf plist :year) → nil
? (setf (getf plist :year) 1961) → 1961
? (remf plist :title) → t
? plist → (:artist "Elvis" :year 1961)
```

- `getf` and `remf` always test using `eq` (not allowing `:test` argument);
- restricts what we can use as keys (typically symbols / keywords).
- Association lists (alists) are more flexible.

# Alists (Association Lists)

- An association list is a list of pairs of keys and values:

```
? (defparameter alist (pairlis '(:artist :title)
                               '("Elvis" "Blue Hawaii")))
→ ((:artist . "Elvis") (:title . "Blue Hawaii"))

? (assoc :artist alist) → (:artist . "Elvis")

? (setf alist (acons :year 1961 alist))
→ ((:artist . "Elvis") (:title . "Blue Hawaii") (:year . 1961))
```

- Note: The result of `cons`'ing something to an atomic value other than `nil` is displayed as a *dotted pair*; `(cons 'a 'b)` → `(a . b)`
- With the `:test` keyword argument we can specify the lookup test function used by `assoc`; keys can be any data type.
- With look-up in a plist or alist, in the worst case, every element in the list has to be searched (linear complexity in list length).

# Hash tables

- While lists are inefficient for indexing large data sets, and arrays restricted to numeric keys, hash tables efficiently handle a large number of (almost) arbitrary type keys.
- Any of the four built-in equality tests can be used for key comparison.

```
? (defparameter table (make-hash-table :test #'equal))
? (gethash "foo" table) → nil
? (setf (gethash "foo" table) 42) → 42
```

- 'Trick' to test, insert and update in one go (specifying 0 as the default):

```
? (incf (gethash "bar" table 0)) → 1
? (gethash "bar" table) → 1
```

- Hash table iteration: use `maphash` or specialized `loop` directives.

# Structures ('Structs')

- defstruct creates a *new abstract data type* with *named slots*.
- Encapsulates a group of related data (i.e. an 'object').
- Each structure type is a new type distinct from all existing Lisp types.
- Defines a new *constructor*, *slot accessors*, and a *type predicate*.

```
? (defstruct album
    (artist "unknown")
    (title "unknown"))

? (defparameter foo (make-album :artist "Elvis"))
  → #S(album :artist "Elvis" :title "unknown")

? (listp foo) → nil

? (album-p foo) → t

? (setf (album-title foo) "Blue Hawaii")

? foo → #S(album :artist "Elvis" :title "Blue Hawaii")
```

# Finally: Give us Those Bonus Points

## Rules of the Game

- Up to four bonus points towards completion of Obligatory Exercise (1).
- Get one post-it; at the top, write down your first and last name.
- Further, write down your UiO account name (e.g. oe, in my case).
- Write each answer on a line of its own, prefix by question number.
- Do not consult with your neighbors; they will likely mess things up.

## After the Quiz

- Post your answers at the front of your table, we will collect all notes.
- Discuss your answers with your neighbor(s); explain why you are right.

# Question (1): Use of `cons` Cells

```
(defparameter foo '(:foo 47 :bar 11))

(defparameter bar '((:foo . 47) (:bar . 11)))
```

**(1) How many `cons` cells are used by `foo` and `bar`, respectively?**

# Question (2): Parameter List Flexibility

```
(defparameter a 47)

(defun foo (a &optional (b 42) c &rest list)
  (list a b c list))

? (foo 'a :b 11 :rest 'list) →
```

**(2) What is the return value of the function call to `foo`?**

# Question (3): Global and Local Variables

```
(defparameter foo '(1 2 3))

(defun foo (foo bar)
  (let ((foo (* foo 2))
        (bar (+ foo 1)))
    (list foo bar)))

? (foo (first (rest foo)) (first (last foo))) →
```

**(3) What is the return value of the function call to `foo`?**

# Question (4): Mysterious Common Lisp

```
(defun ? (?)
  (if (null ?)
    ?
    (cons (first ?) (? (rest ?)))))
```

**(4) What argument type does ? take, and what does it compute?**

# Answer (1): Use of `cons` Cells

```
(defparameter foo '(:foo 47 :bar 11))

(defparameter bar '((:foo . 47) (:bar . 11)))
```

**(1) How many `cons` cells are used by `foo` and `bar`, respectively?**

4 (in both cases)

# Answer (2): Parameter List Flexibility

```
(defparameter a 47)

(defun foo (a &optional (b 42) c &rest list)
  (list a b c list))

? (foo 'a :b 11 :rest 'list) →
```

**(2) What is the return value of the function call to `foo`?**

`(a :b 11 (:rest list))`

# Answer (3): Global and Local Variables

```
(defparameter foo '(1 2 3))

(defun foo (foo bar)
  (let ((foo (* foo 2))
        (bar (+ foo 1)))
    (list foo bar)))

? (foo (first (rest foo)) (first (last foo))) →
```

**(3) What is the return value of the function call to `foo`?**

`(4 3)`

# Answer (4): Mysterious Common Lisp

```
(defun ? (?)
  (if (null ?)
    ?
    (cons (first ?) (? (rest ?)))))
```

**(4) What argument type does ? take, and what does it compute?**

Lists; `foo` returns a fresh, equivalent copy.

# Good Lisp Style

## Bottom-Up Design

- Instead of trying to solve everything with one large function: Build your program with layers of smaller functions.
  - Eliminate repetition and patterns.
- Related; define *abstraction barriers*.
  - Separate the code that uses a given data abstraction from the code that implements that data abstraction.
- Promotes code re-use:
  - Makes the code shorter and easier to read, debug, and maintain.

- Somewhat more mundane:
  - Adhere to the time-honored *80 column rule*.
  - Close multiple parentheses on the same line.
  - Use auto-indentation (`TAB`) in emacs.

# Next Time: Wednesday, September 17, 2014

- Can we automatically infer the meaning of words?
- Distributional semantics
- Vector spaces: Spatial models for representing data
- Semantic spaces