— INF4820 —
Algorithms for AI and NLP

More Common Lisp

Erik Velldal & Stephan Oepen

Language Technology Group (LTG)

September 2, 2015

Agenda

Previous lecture

- Common Lisp essentials
- S-expressions (= atoms + lists of s-expressions)
- Recursion
- Quote
- List processing

Agenda

Previous lecture

- Common Lisp essentials
- S-expressions (= atoms + lists of s-expressions)
- Recursion
- Quote
- List processing

Today

- More Common Lisp
- Higher-order functions
- Iteration and `loop`
- More data structures (alists, arrays, hash-tables, structs, and more)

Higher-order functions

- = functions taking other functions as arguments or return values.

Higher-order functions

- = functions taking other functions as arguments or return values.

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

Higher-order functions

- = functions taking other functions as arguments or return values.

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

Higher-order functions

- = functions taking other functions as arguments or return values.

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

? (filter '(11 22 33 44 55) #'evenp)
→ (22 44)

? (filter '(11 22 33 44 55) #'oddp)
→ (11 33 55)
```

Higher-order functions

- = functions taking other functions as arguments or return values.

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

? (filter '(11 22 33 44 55) #'evenp)
→ (22 44)

? (filter '(11 22 33 44 55) #'oddp)
→ (11 33 55)
```

Higher-order functions

- = functions taking other functions as arguments or return values.

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

? (filter '(11 22 33 44 55) #'evenp)
→ (22 44)

? (filter '(11 22 33 44 55) #'oddp)
→ (11 33 55)
```

Higher-order functions

- = functions taking other functions as arguments or return values.

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

? (filter '(11 22 33 44 55) #'evenp)
→ (22 44)

? (filter '(11 22 33 44 55) #'oddp)
→ (11 33 55)
```

Higher-order functions

- = functions taking other functions as arguments or return values.

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

? (filter '(11 22 33 44 55) #'evenp)
→ (22 44)

? (filter '(11 22 33 44 55) #'oddp)
→ (11 33 55)
```

Higher-order functions

- = functions taking other functions as arguments or return values.

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

? (filter '(11 22 33 44 55) #'evenp)
→ (22 44)

? (filter '(11 22 33 44 55) #'oddp)
→ (11 33 55)
```

Anonymous functions

- We can create functions without naming them with `defun`:

```
(lambda (parameters) body)
```

instead of

```
(defun name (parameters) body)
```

Anonymous functions

- We can create functions without naming them with `defun`:

```
(lambda (parameters) body)
```

instead of

```
(defun name (parameters) body)
```

- For example:

```
? (filter '(11 22 33 44 55)
          #'(lambda (x)
              (and (> x 20)
                   (< x 50))))

→ (22 33 44)
```

Returning functions

- Having a function return another function is easy:
- Make the return value a `lambda` expression.

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

Returning functions

- Having a function return another function is easy:
- Make the return value a `lambda` expression.

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

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

? (filter foo (create-range-test 10 30))
→ (11 22)
```

Returning functions

- Having a function return another function is easy:
- Make the return value a lambda expression.

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

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

? (filter foo (create-range-test 10 30))
→ (11 22)

? (filter foo (create-range-test 20 50))
→ (22 33 44)
```

Parameter lists: Variable arities and ordering

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)
```

Parameter lists: Variable arities and ordering

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)
```

Parameter lists: Variable arities and ordering

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
```

Macros

- 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've already used some built-in Common Lisp macros: `and`, `or`, `if`, `cond`, `defun`, `setf`, etc.

Macros

- 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've already used some built-in Common Lisp macros: `and`, `or`, `if`, `cond`, `defun`, `setf`, etc.
- Although macro writing is out of the scope of this course, let's 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.

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

→ (0 2 4)
```

Iteration

- While recursion is a powerful control structure,
- sometimes *iteration* comes more natural.
- `dolist` and `dotimes` are fine for simple iteration.

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

→ (0 2 4)
```

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

→ (0 2 4)
```

Iteration

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

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

→ (0 2 4)
```

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

→ (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.

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: 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: 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: a few more examples

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

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: 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: 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 1 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))
```

`loop`: The Swiss Army Knife of Iteration

- Iteration over lists or vectors: `for` *symbol* { `in` | `on` | `across` } *sequence*
- 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`.

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
```

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.

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)))
```

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.

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 '(1 2 3) '(1 2 3)) → nil
```

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 '(1 2 3) '(1 2 3)) → nil

? (equal '(1 2 3) '(1 2 3)) → t
```

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 '(1 2 3) '(1 2 3)) → nil

? (equal '(1 2 3) '(1 2 3)) → t

? (eq 42 42) → ? [implementation-dependent]
```

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 '(1 2 3) '(1 2 3)) → nil

? (equal '(1 2 3) '(1 2 3)) → t

? (eq 42 42) → ? [implementation-dependent]

? (eql 42 42) → t
```

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 '(1 2 3) '(1 2 3)) → nil

? (equal '(1 2 3) '(1 2 3)) → t

? (eq 42 42) → ? [implementation-dependent]

? (eql 42 42) → t

? (eql 42 42.0) → nil
```

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 '(1 2 3) '(1 2 3)) → nil
? (equal '(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
```

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 '(1 2 3) '(1 2 3)) → nil
? (equal '(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
```

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 '(1 2 3) '(1 2 3)) → nil
? (equal '(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
```

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 '(1 2 3) '(1 2 3)) → nil
? (equal '(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.

…now we'll do a quick tour of

some other

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)
```

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.

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 ‘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.:

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
```

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
```

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))) →
```

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
```

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"
```

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"
```

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)
```

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))) →
```

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
```

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)
```

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)
```

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

Sequence functions and keyword parameters

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

Sequence functions and keyword parameters

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

```
? (member "bar" '("foo" "bar" "baz"))
  →
```

Sequence functions and keyword parameters

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

```
? (member "bar" '("foo" "bar" "baz"))
  → nil
```

Sequence functions and keyword parameters

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

```
? (member "bar" '("foo" "bar" "baz"))
  → nil

? (member "bar" '("foo" "bar" "baz") :test #'equal)
  →
```

Sequence functions and keyword parameters

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

```
? (member "bar" '("foo" "bar" "baz"))
  → nil

? (member "bar" '("foo" "bar" "baz") :test #'equal)
  → ("bar" "baz")
```

Sequence functions and keyword parameters

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

```
? (member "bar" '("foo" "bar" "baz"))
  → nil

? (member "bar" '("foo" "bar" "baz") :test #'equal)
  → ("bar" "baz")

? (defparameter bar '(("baz" 23) ("bar" 47) ("foo" 11)))

? (sort bar #'< :key #'(lambda (foo) (first (rest foo))))
  →
```

Sequence functions and keyword parameters

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

```
? (member "bar" '("foo" "bar" "baz"))
  → nil

? (member "bar" '("foo" "bar" "baz") :test #'equal)
  → ("bar" "baz")

? (defparameter bar '(("baz" 23) ("bar" 47) ("foo" 11)))

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

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"))
```

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"
```

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
```

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
```

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
```

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)
```

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"))
```

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")
```

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))
```

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)`

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.

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))
```

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
```

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
```

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 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"))
```

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"))
```

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")
```

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
```

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")
```

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")
```

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 implement that data abstraction.
- Promotes code re-use:
  - Makes the code shorter and easier to read, debug and maintain.

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 implement 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 parens on the same line.
  - Use Emacs' auto-indentation (`TAB`).

Next week

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