Lisp Users and Vendors Conference

August 10, 1993

Tutorial on Good Lisp Programming Style

Peter Norvig

Sun Microsystems Labs Inc.

Kent Pitman

Harlequin, Inc.

Outline

Good Lisp Programming Style

"Elegance is not optional." – *Richard A. O'Keefe*

Good style (in any language) leads to programs that are:

- Understandable
- Reusable
- Extensible
- Efficient
- Easy to develop/debug

It also helps correctness, robustness, compatibility
Our maxims of good style are:

- Be explicit
- Be specific
- Be concise
- Be consistent
- Be helpful (anticipate the reader's needs)
- Be conventional (don't be obscure)
- Build abstractions at a usable level
- Allow tools to interact (referential transparency)

Good style is the "underware" that supports a program

What To Believe

Don't believe everything we tell you. (Just most.)

Worry less about **what** to believe and more about **why**. Know where your "Style Rules" come from:

- **Religion, Good vs. Evil** "This way is better."
- **Philosophy** "This is consistent with other things."
- **Robustness, Liability, Safety, Ethics** "I'll put in redundant checks to avoid something horrible."
- **Legality** "Our lawyers say do it this way."
- **Personality, Opinion** "**I** like it this way."
- **Compatibility** "Another tool expects this way."
- **Portability** "Other compilers prefer this way."
- **Cooperation, Convention** "It has to be done some uniform way, so we agreed on this one."
- **Habit, Tradition** "We've always done it this way."
- **Ability** "My programmers aren't sophisticated enough."
- **Memory** "Knowing how I *would* do it means I don't have to remember how I *did* do it."
- **Superstition** "I'm scared to do it differently."
- **Practicality** "This makes other things easier."

It's All About Communication

Expression + Understanding = Communication

Programs communicate with:

- Human readers
- Compilers
- Text editors (arglist, doc string, indent)
- Tools (trace, step, apropos, xref, manual)
- Users of the program (indirect communication)

Know the Context

When reading code:

- Know **who** wrote it and **when**.

When writing code:

- Annotate it with comments.
- **Sign** and **date** your comments!
  (Should be an editor command to do this)

Some things to notice:

- People's style changes over time.
- The same person at different times can seem like a different person.
- Sometimes that person is you.

Value Systems Are Not Absolute

Style rules cannot be viewed in isolation.
They often overlap in conflicting ways.

The fact that style rules conflict with one another reflects the natural fact that real-world goals conflict. A good programmer makes trade-offs in programming style that reflect underlying priority choices among various major goals:

- Understandable
- Reusable
- Extensible
- Efficient (coding, space, speed, ...)
- Easy to develop/debug

Why Good Style is Good

Good style helps build the current program, and the next one:

- Organizes a program, relieving human memory needs
- Encourages modular, reusable parts

Style is not just added at the end. It plays a part in:

- Organization of the program into files
- Top-level design, structure and layout of each file
- Decomposition into modules and components
- Data-structure choice
- Individual function design/implementation
- Naming, formatting, and documenting standards

Why Style is Practical: Memory

"When I was young, I could imagine a castle with twenty rooms with each room having ten different objects in it. I would have no problem. I can't do that anymore. Now I think more in terms of earlier experiences. I see a network of inchoate clouds, instead of the picture-postcard clearness. But I do write better programs." – *Charles Simonyi*

"Some people are good programmers because they can handle many more details than most people. But there are a lot of disadvantages in selecting programmers for that reason—it can result in programs no on else can maintain." – *Butler Lampson*

"Pick out any three lines in my program, and I can tell you where they're from and what they do." – *David McDonald*

Good style replaces the need for great memory:

- Make sure any 3 (5? 10?) lines are self-explanatory
  Also called "referential transparency"
  Package complexity into objects and abstractions; not global variables/dependencies
- Make it "fractally" self-organizing all the way up/down
- Say what you mean
- Mean what you say

Why Style is Practical: Reuse

Structured Programming encourages modules that meet specifications and can be reused within the bounds of that specification.

Stratified Design encourages modules with commonly-needed functionality, which can be reused even when the specification changes, or in another program.

Object-Oriented Design is stratified design that concentrates on classes of objects and on information hiding.

You should aim to reuse:

- Data types (classes)
- Functions (methods)
- Control abstractions
- Interface abstractions (packages, modules)
- Syntactic abstractions (macros and whole languages)

Say What You Mean

"Say what you mean, simply and directly." – *Kernighan & Plauger*

Say what you mean in **data** (be specific, concise):

- Use data abstractions
- Define languages for data, if needed
- Choose names wisely

Say what you mean in **code** (be concise, conventional):

- Define interfaces clearly
- Use Macros and languages appropriately
- Use built-in functions
- Create your own abstractions
- Don't do it twice if you can do it once

In **annotations** (be explicit, helpful):

- Use appropriate detail for comments
- Documentation strings are better than comments
- Say what it is for, not just what it does
- Declarations and assertions
- Systems (and test files, etc.)

Be Explicit

Optional and Keyword arguments.

If you have to look up the default value, you need to supply it. You should only take the default if you truly believe you don't care or if you're sure the default is well-understood and well-accepted by all.

For example, when opening a file, you should almost never consider omitting the `:direction` keyword argument, even though you know it will default to `:input`.

Declarations.

If you know type information, declare it. Don't do what some people do and only declare things you know the compiler will use. Compilers change, and you want your program to naturally take advantage of those changes without the need for ongoing intervention.

Also, declarations are for communication with human readers, too—not just compilers.

Comments.

If you're thinking of something useful that others might want to know when they read your code and that might not be instantly apparent to them, make it a comment.

Be Specific

Be as specific as your data abstractions warrant,
but no more.

Choose:

```
;; more specific             ;; more abstract
(mapc #'process-word         (map nil #'process-word
      (first sentences))          (elt sentences 0))
```

Most specific conditional:

- `if` for two-branch expression
- `when`, `unless` for one-branch statement
- `and`, `or` for boolean value only
- `cond` for multi-branch statement or expression

```
;; Violates Expectation:     ;; Follows Expectation:
(and (numberp x) (cos x))    (and (numberp x) (> x 3))
(if (numberp x) (cos x))     (if (numberp x) (cos x) nil)
(if (numberp x) (print x))   (when (numberp x) (print x))
```

Be Concise

Test for the simplest case. If you make the same test (or return the same result) in two places, there must be an easier way.

Bad: *verbose, convoluted*

```
(defun count-all-numbers (alist)
       (cond
           ((null alist) 0)
           (t (+ (if (listp (first alist))
                         (count-all-numbers (first alist))
                      (if (numberp (first alist)) 1 0))
                 (count-all-numbers (rest alist)) )) ))
```

- Returns 0 twice
- Nonstandard indentation
- `alist` suggests association list

Good:

```
(defun count-all-numbers (exp)
  (typecase exp
    (cons   (+ (count-all-numbers (first exp))
               (count-all-numbers (rest exp))))
    (number 1)
    (t      0)))
```

`cond` instead of `typecase` is equally good (less specific, more conventional, consistent).

Be Concise

Maximize LOCNW: lines of code not written.
"Shorter is better and shortest is best."
– *Jim Meehan*

Bad: *too verbose, inefficient*

```
(defun vector-add (x y)
  (let ((z nil) n)
    (setq n (min (list-length x) (list-length y)))
    (dotimes (j n (reverse z))
      (setq z (cons (+ (nth j x) (nth j y)) z)))))

(defun matrix-add (A B)
  (let ((C nil) m)
    (setq m (min (list-length A) (list-length B)))
    (dotimes (i m (reverse C))
      (setq C (cons (vector-add (nth i A)
                                (nth i B)) C)))))
```

- Use of `nth` makes this $O(n^2)$
- Why `list-length`? Why not `length` or `mapcar`?
- Why not `nreverse`?
- Why not use arrays to implement arrays?
- The return value is hidden

Be Concise

Better: *more concise*

```
(defun vector-add (x y)
  "Element-wise add of two vectors"
  (mapcar #'+ x y))

(defun matrix-add (A B)
  "Element-wise add of two matrices (lists of lists)"
  (mapcar #'vector-add A B))
```

Or use generic functions:

```
(defun add (&rest args)
  "Generic addition"
  (if (null args)
      0
      (reduce #'binary-add args)))

(defmethod binary-add ((x number) (y number))
  (+ x y))

(defmethod binary-add ((x sequence) (y sequence))
  (map (type-of x) #'binary-add x y))
```

Be Helpful

Documentation should be organized around tasks the user needs to do, not around what your program happens to provide. Adding documentation strings to each function usually doesn't tell the reader how to use your program, but hints in the right place can be very effective.

Good: *(from Gnu Emacs online help)*

`next-line:` Move cursor vertically down ARG lines. ...If you are thinking of using this in a Lisp program, consider using 'forward-line' instead. It is usually easier to use and more reliable (no dependence on goal column, etc.).

`defun:` defines NAME as a function. The definition is `(lambda ARGLIST [DOCSTRING] BODY...)`. See also the function `interactive`.

These anticipate user's use and problems.

Be Conventional

Build your own functionality to parallel existing features
Obey naming conventions:
`with-`*something*, `do`*something* macros

Use built-in functionality when possible

- Conventional: reader will know what you mean
- Concise: reader doesn't have to parse the code
- Efficient: has been worked on heavily

Bad: *non-conventional*

```
(defun add-to-list (elt list)
  (cond ((member elt lst) lst)
        (t (cons elt lst))))
```

Good: *use a built-in function*

(left as an exercise)

"Use library functions" – *Kernighan & Plauger*

Be Consistent

Some pairs of operators have overlapping capabilities. Be consistent about which you use in neutral cases (where either can be used), so that it is apparent when you're doing something unusual.

Here are examples involving `let` and `let*`. The first exploits parallel binding and the second sequential. The third is neutral.

```
(let ((a b) (b a)) ...)

(let* ((a b) (b (* 2 a)) (c (+ b 1))) ...)

(let ((a (* x (+ 2 y))) (b (* y (+ 2 x)))) ...)
```

Here are analogous examples using `flet` and `labels`. The first exploits closure over the local function, the second exploits non-closure. The third is neutral.

```
(labels ((process (x) ... (process (cdr x)) ...)) ...)

(flet ((foo (x) (+ (foo x) 1))) ...)

(flet ((add3 (x) (+ x 3))) ...)
```

In both cases, you could choose things the other way around, always using `let*` or `labels` in the neutral case, and `let` or `flet` in the unusual case. Consistency matters more than the actual choice. Most people, however, think of `let` and `flet` as the normal choices.

Choose the Right Language

Choose the appropriate language, and use appropriate features in the language you choose. Lisp is not the right language for every problem.

"You got to dance with the one that brung you."
– *Bear Bryant*

Lisp is good for:

- Exploratory programming
- Rapid prototyping
- Minimizing time-to-market
- Single-programmer (or single-digit team) projects
- Source-to-source or data-to-data transformation
  Compilers and other translators
  Problem-specific languages
- Dynamic dispatch and creation
  (compiler available at run-time)
- Tight integration of modules in one image
  (as opposed to Unix's character pipe model)
- High degree of interaction (read-eval-print, CLIM)
- User-extensible applications (gnu emacs)

"I believe good software is written by small teams of two, three, or four people interacting with each other at a very high, dense level." – *John Warnock*

Choose the Right Language

"Once you are an experienced Lisp programmer, it's hard to return to any other language." – *Robert R. Kessler*

Current Lisp implementations are not so good for:

- Persistent storage (data base)
- Maximizing resource use on small machines
- Projects with hundreds of programmers
- Close communication with foreign code
- Delivering small-image applications
- Real-time control (but Gensym did it)
- Projects with inexperienced Lisp programmers
- Some kinds of numerical or character computation (Works fine with careful declarations, but the Lisp efficiency model is hard to learn.)

Built-in Functionality

"No doubt about it, Common Lisp is a big language"
– *Guy Steele*

- 622 built-in functions (in one pre-ANSI CL)
- 86 macros
- 27 special forms
- 54 variables
- 62 constants

But what counts as the language itself?

- C++ has 48 reserved words
- ANSI CL is down to 25 special forms
- The rest can be thought of as a required library

Either way, the Lisp programmer needs some help:
Which built-in functionality to make use of
How to use it

DEFVAR and DEFPARAMETER

Use **defvar** for things you don't want to re-initialize upon re-load.

```
(defvar *options* '())
(defun add-option (x) (pushnew x *options*))
```

Here you might have done (**add-option ...**) many times before you re-load the file—perhaps some even from another file. You usually don't want to throw away all that data just because you re-load this definition.

On the other hand, some kinds of options do want to get re-initialized upon re-load...

```
(defparameter *use-experimental-mode* nil
  "Set this to T when experimental code works.")
```

Later you might edit this file and set the variable to T, and then re-load it, wanting to see the effect of your edits.

Recommendation: Ignore the part in CLtL that says **defvar** is for variables and **defparameter** is for parameters. The only useful difference between these is that **defvar** does its assignment only if the variable is unbound, while **defparameter** does its assignment unconditionally.

EVAL-WHEN

```
(eval-when (:execute) ...)

=

(eval-when (:compile-toplevel) ...)
+
(eval-when (:load-toplevel) ...)
```

Also, take care about explicitly nesting `eval-when` forms. The effect is *not* generally intuitive for most people.

FLET to Avoid Code Duplication

Consider the following example's duplicated use of `(f (g (h)))`.

```
(do ((x (f (g (h)))
        (f (g (h)))))
    (nil) ...)
```

Every time you edit one of the `(f (g (h)))`'s, you probaby want to edit the other, too. Here is a better modularity:

```
(flet ((fgh () (f (g (h)))))
  (do ((x (fgh) (fgh))) (nil) ...))
```

(This might be used as an argument against `do`.)

Similarly, you might use local functions to avoid duplication in code branches that differ only in their dynamic state. For example,

```
(defmacro handler-case-if (test form &rest cases)
  (let ((do-it (gensym "DO-IT")))
    `(flet ((,do-it () ,form))
       (if test
           (handler-case (,do-it) ,@cases)
           (,do-it)))))
```

DEFPACKAGE

Programming in the large is supported by a design style that separates code into modules with clearly defined interfaces.

The Common Lisp package system serves to avoid name clashes between modules, and to define the interface to each module.

- There is no top level (be thread-safe)
- There are other programs (use packages)
- Make it easy for your consumers
  Export only what the consumer needs
- Make it easy for maintainers
  License to change non-exported part

```
(defpackage "PARSER"
  (:use "LISP" #+Lucid "LCL" #+Allegro "EXCL")
  (:export "PARSE" "PARSE-FILE" "START-PARSER-WINDOW"
           "DEFINE-GRAMMAR" "DEFINE-TOKENIZER"))
```

Some put exported symbols at the top of the file where they are defined.

We feel it is better to put them in the `defpackage`, and use the editor to find the corresponding definitions.

Understanding Conditions vs Errors

Lisp assures that most errors in code will not corrupt data by providing an active condition system.

Learn the difference between **errors** and **conditions**. All errors are conditions; not all conditions are errors.

Distinguish three concepts:

- Signaling a condition—
  Detecting that something unusual has happened.
- Providing a restart—
  Establishing one of possibly several options for continuing.
- Handling a condition—
  Selecting how to proceed from available options.

Error Detection

Pick a level of error detection and handling that matches your intent. Usually you don't want to let bad data go by, but in many cases you also don't want to be in the debugger for inconsequential reasons.

Strike a balance between tolerance and pickiness that is appropriate to your application.

Bad: *what if its not an integer?*

```
(defun parse-date (string)
  "Read a date from a string. ..."
  (multiple-value-bind (day-of-month string-position)
      (parse-integer string :junk-allowed t)
    ...))
```

Questionable: *what if memory runs out?*

```
(ignore-errors (parse-date string))
```

Better: *catches expected errors only*

```
 (handler-case (parse-date string)
   (parse-error nil))
```

Writing Good Error Messages

- Use full sentences in error messages (uppercase initial, trailing period).
- No `"Error: "` or `";;"` prefix. The system will supply such a prefix if needed.
- Do not begin an error message with a request for a fresh line. The system will do this automatically if necessary.
- As with other format strings, don't use embedded tab characters.
- Don't mention the consequences in the error message. Just describe the situation itself.
- Don't presuppose the debugger's user interface in describing how to continue. This may cause portability problems since different implementations use different interfaces. Just describe the abstract effect of a given action.
- Specify enough detail in the message to distinguish it from other errors, and if you can, enough to help you debug the problem later if it happens.

Writing Good Error Messages (cont'd)

Bad:

```
(error "~%>> Error: Foo. Type :C to continue.")
```

Better:

```
(cerror "Specify a replacement sentence interactively."
        "An ill-formed sentence was encountered:~% ~A"
        sentence)
```

Using the Condition System

Start with these:

- error, cerror
- warn
- handler-case
- with-simple-restart
- unwind-protect

Good: *standard use of warn*

```
(defvar *word* '?? "The word we are currently working on.")

(defun lex-warn (format-str &rest args)
  "Lexical warning; like warn, but first tells what word
  caused the warning."
  (warn "For word ~a: ~?" *word* format-str args))
```

HANDLER-CASE, WITH-SIMPLE-RESTART

Good: *handle specific errors*

```
(defun eval-exp (exp)
  "If possible evaluate this exp; otherwise return it."
  ;; Guard against errors in evaluating exp
  (handler-case
    (if (and (fboundp (op exp))
             (every #'is-constant (args exp)))
        (eval exp)
        exp)
    (arithmetic-error () exp)))
```

Good: *provide restarts*

```
(defun top-level (&key (prompt "=> ") (read #'read)
                       (eval #'eval) (print #'print))
  "A read-eval-print loop."
  (with-simple-restart
      (abort "Exit out of the top level.")
    (loop
     (with-simple-restart
         (abort "Return to top level loop.")
       (format t "~&~a" prompt)
       (funcall print (funcall eval (funcall read)))))))
```

UNWIND-PROTECT

`unwind-protect` implements important functionality that everyone should know how to use. It is *not* just for system programmers.

Watch out for multi-tasking, though. For example, implementing some kinds of state-binding with `unwind-protect` might work well in a single-threaded environment, but in an environment with multi-tasking, you often have to be a little more careful.

(**unwind-protect** (**progn** *form*$_1$ *form*$_2$ ... *form*$_n$)
  cleanup$_1$ *cleanup*$_2$ ... *cleanup*$_n$)

- Never assume **form**$_1$ will get run at all.
- Never assume **form**$_n$ won't run to completion.

UNWIND-PROTECT (cont'd)

Often you need to save state before entering the `unwind-protect` and test before you restore state:

Possibly Bad: *(with multi-tasking)*

```
(catch 'robot-op
  (unwind-protect
    (progn (turn-on-motor)
           (manipulate) )
    (turn-off-motor)))
```

Good: *(safer)*

```
(catch 'robot-op
  (let ((status (motor-status motor)))
    (unwind-protect
         (progn (turn-on-motor motor)
                (manipulate motor))
      (when (motor-on? motor)
        (turn-off-motor motor))
      (setf (motor-status motor) status))))
```

I/O Issues: Using FORMAT

- Don't use Tab characters in format strings (or any strings intended for output). Depending on what column your output starts in, the tab stops may not line up the same on output as they did in the code!
- Don't use `"#<~S ~A>"` to print unreadable objects. Use `print-unreadable-object` instead.
- Consider putting format directives in uppercase to make them stand out from lowercase text surrounding.
  For example, `"Foo:  ~A"` instead of `"Foo:  ~a"`.
- Learn useful idioms. For example: `~{~A~^, ~}` and `~:p`.
- Be conscious of when to use `~&` versus `~%`.
  Also, `"~2%"` and `"~2&"` are also handy.

  Most code which outputs a single line should start with `~&` and end with `~%`.

  ```
  (format t "~&This is a test.~%")
  This is a test.
  ```
- Be aware of implementation extensions. They may not be portable, but for non-portable code might be very useful. For example, Genera's $\rightarrow$ and $\leftarrow$ for handling indentation.

Using Streams Correctly

- `*standard-output*` and `*standard-input*` vs `*terminal-io*`

  Do not assume `*standard-input*` and `*standard-output*` will be bound to `*terminal-io*` (or, in fact, to any interactive stream). You can bind them to such a stream, however.

  Try not to use `*terminal-io*` directly for input or output. It is primarily available as a stream to which other streams may be bound, or may indirect (*e.g.,* by synonym streams).

- `*error-output*` vs `*debug-io*`

  Use `*error-output*` for warnings and error messages that are not accompanied by any user interaction.

  Use `*debug-io*` for interactive warnings, error messages, and other interactions not related to the normal function of a program.

  In particular, do not first print a message on `*error-output*` and then do a debugging session on `*debug-io*`, expecting those to be the same stream. Instead, do each interaction consistently on one stream.

Using Streams Correctly (cont'd)

- `*trace-output*`

  This can be used for more than just receiving the output of `trace`. If you write debugging routines that conditionally print helpful information without stopping your running program, consider doing output to this stream so that if `*trace-output*` is redirected, your debugging output will too.

A useful test: If someone re-bound only one of several I/O streams you are using, would it make your output look stupid?

Using Near-Standard Tools

Some functionality is not built in to the language, but is used by most programmers. This divides into extensions to the language and tools that help you develop programs.

Extensions

- `defsystem` to define a program
- `CLIM`, `CLX`, etc. graphics libraries

Tools

- `emacs` from FSF, Lucid
  indentation, font/color support
  definition/arglist/doc/regexp finding
  communication with lisp
- `xref`, `manual`, etc. from CMU
- Browsers, debuggers, profilers from vendors

DEFSYSTEM

Pick a public domain version of **defsystem** (unfortunately, dpANS CL has no standard).

- Put absolute pathnames in one place only
- Load everything through the defsystem
- Distinguish compiling from loading
- Optionally do version control

```
(defpackage "PARSER" ...)

(defsystem parser
  (:source "/lab/indexing/parser/*")
  (:parts utilities "macros" "grammar" "tokenizer"
          "optimizer" "debugger" "toplevel"
          #+CLIM "clim-graphics" #+CLX "clx-graphics"))
```

- Make sure your system loads with no compiler warnings
  (first time and subsequent times)
  (learn to use `(declare (ignore ...))`)
- Make sure the system can be compiled from scratch
  (eliminate lingering bootstrapping problems)

Editor Commands

Your editor should be able to do the following:

- Move about by s-expressions and show matching parens
- Indent code properly
- Find unbalanced parens
- Adorn code with fonts and colors
- Find the definition of any symbol
- Find arguments or documentation for any symbol
- Macroexpand any expression
- Send the current expression, region or file to Lisp to be evaluated or compiled
- Keep a history of commands sent to Lisp and allow you to edit and resend them
- Work with keyboard, mouse, and menus

Emacs can do all these things. If your editor can't, complain until it is fixed, or get a new one.

Emacs: Indentation and Comments

Don't try to indent yourself.
Instead, let the editor do it.
A near-standard form has evolved.

- 80-column maximum width
- Obey comment conventions
  ; for inline comment
  ;; for in-function comment
  ;;; for between-function comment
  ;;;; for section header (for outline mode)
- `cl-indent` library can be told how to indent
  `(put 'defvar 'common-lisp-indent-function '(4 2 2))`
- `lemacs` can provide fonts, color

```
(hilit::modes-list-update "Lisp"
  '((";;.*" nil hilit2) ...))
```

Abstraction

All programming languages allow the programmer to define *abstractions*. All modern languages provide support for:

- Data Abstraction (abstract data types)
- Functional Abstraction (functions, procedures)

Lisp and other languages with closures (*e.g.*, ML, Sather) support:

- Control Abstraction (defining iterators and other new flow of control constructs)

Lisp is unique in the degree to which it supports:

- Syntactic Abstraction (macros, whole new languages)

Design: Where Style Begins

"The most important part of writing a program is designing the data structures. The second most important part is breaking the various code pieces down." – *Bill Gates*

"Expert engineers stratify complex designs. ...The parts constructed at each level are used as primitives at the next level. Each level of a stratified design can be thought of as a specialized language with a variety of primitives and means of combination appropriate to that level of detail." – *Harold Abelson and Gerald Sussman*

"Decompose decisions as much as possible. Untangle aspects which are only seemingly independent. Defer those decisions which concern details of representation as long as possible." – *Niklaus Wirth*

Lisp supports all these approaches:

- Data Abstraction: classes, structures, deftype
- Functional Abstraction: functions, methods
- Interface Abstraction: packages, closures
- Object-Oriented: CLOS, closures
- Stratified Design: closures, all of above
- Delayed Decisions: run-time dispatch

Design: Decomposition

"A Lisp procedure is like a paragraph."
– *Deborah Tatar*

"You should be able to explain any module in one sentence." – *Wayne Ratliff*

- Strive for simple designs
- Break the problem into parts
  Design useful subparts (stratified)
  Be opportunistic; use existing tools
- Determine dependencies
  Re-modularize to reduce dependencies
  Design most dependent parts first

We will cover the following kinds of abstraction:

- Data abstraction
- Functional abstraction
- Control abstraction
- Syntactic abstraction

Data Abstraction

Write code in terms of the problem's data types, not the types that happen to be in the implementation.

- Use `defstruct` or `defclass` for record types
- Use inline functions as aliases (not macros)
- Use `deftype`
- Use declarations and `:type` slots for efficiency and/or documentation
- Variable names give informal type information

Pretty Good: *specifies some type info*

```
(defclass event ()
  ((starting-time :type integer)
   (location :type location)
   (duration :type integer :initform 0)))
```

Better: *problem-specific type info*

```
(deftype time () "Time in seconds" 'integer)

(defconstant +the-dawn-of-time+ 0
  "Midnight, January 1, 1900"

(defclass event ()
  ((starting-time :type time :initform +the-dawn-of-time+)
   (location :type location)
   (duration :type time :initform 0)))
```

Use Abstract Data Types

Introduce abstract data types with accessors:

Bad: *obscure accessor, eval*

```
(if (eval (cadar rules)) ...)
```

Better: *introduce names for accessors*

```
(declaim (inline rule-antecedent))
(defun rule-antecedent (rule) (second rule))

(if (holds? (rule-antecedent (first rules))) ...)
```

Usually Best: *introduce first-class data type*

```
(defstruct rule
  name antecedent consequent)
```

or

```
(defstruct (rule (:type list))
  name antecedent consequent)
```

or

```
(defclass rule ()
  (name antecedent consequent))
```

Implement Abstract Data Types

Know how to map from common abstract data types to Lisp implementations.

- Set: list, bit-vector, integer, any table type
- Sequence: list, vector, delayed-evaluation stream
- Stack: list, vector (with fill-pointer)
- Queue: tconc, vector (with fill-pointer)
- Table: hash table, alist, plist, vector
- Tree, Graph: cons, structures, vector, adjacency matrix

Use implementations that are already supported (e.g. **`union, intersection, length`** for sets as lists; **`logior, logand, logcount`** for sets as integers.

Don't be afraid to build a new implementation if profiling reveals a bottleneck. (If Common Lisp's hash tables are too inefficient for your application, consider building a specialized hash table in Lisp before you build a specialized hash table in C.)

Inherit from Data Types

Reuse by inheritance as well as direct use

- structures support single inheritance
- classes support multiple inheritance
- both allow some over-riding
- classes support mixins

Consider a class or structure for the whole program

- Eliminates clutter of global variables
- Thread-safe
- Can be inherited and modified

Functional Abstraction

Every function should have:

- A single specific purpose
- If possible, a generally useful purpose
- A meaningful name
  (names like `recurse-aux` indicate problems)
- A structure that is simple to understand
- An interface that is simple yet general enough
- As few dependencies as possible
- A documentation string

Decomposition

Decompose an algorithm into functions that are simple, meaningful and useful.

Example from comp.lang.lisp discussion of `loop` vs. `map`:

```
(defun least-common-superclass (instances)
  (let ((candidates
         (reduce #'intersection
                 (mapcar #'(lambda (instance)
                             (clos:class-precedence-list
                              (class-of instance)))
                         instances)))
        (best-candidate (find-class t)))
    (mapl
     #'(lambda (candidates)
         (let ((current-candidate (first candidates))
               (remaining-candidates (rest candidates)))
           (when (and (subtypep current-candidate
                                best-candidate)
                      (every
                       #'(lambda (remaining-candidate)
                           (subtypep current-candidate
                                     remaining-candidate))
                       remaining-candidates))
             (setf best-candidate current-candidate))))
     candidates)
    best-candidate))
```

Decomposition

Very Good: *Chris Riesbeck*

```
(defun least-common-superclass (instances)
  (reduce #'more-specific-class
          (common-superclasses instances)
          :initial-value (find-class 't)))

(defun common-superclasses (instances)
  (reduce #'intersection
          (superclass-lists instances)))

(defun superclass-lists (instances)
  (loop for instance in instances
        collect (clos:class-precedence-list
                  (class-of instance))))

(defun more-specific-class (class1 class2)
  (if (subtypep class2 class1) class2 class1))
```

- Each function is very understandable
- Control structure is clear:
  Two reduces, an intersection and a loop/collect
- But reusablity is fairly low

Decomposition

Equally Good: *and more reusable*

```
(defun least-common-superclass (instances)
  "Find a least class that all instances belong to."
  (least-upper-bound (mapcar #'class-of instances)
                     #'clos:class-precedence-list
                     #'subtypep))

(defun least-upper-bound (elements supers sub?)
  "Element of lattice that is a super of all elements."
  (reduce #'(lambda (x y)
              (binary-least-upper-bound x y supers sub?))
          elements))

(defun binary-least-upper-bound (x y supers sub?)
  "Least upper bound of two elements."
  (reduce-if sub? (intersection (funcall supers x)
                                (funcall supers y))))

(defun reduce-if (pred sequence)
  "E.g. (reduce-if #'> numbers) computes maximum"
  (reduce #'(lambda (x y) (if (funcall pred x y) x y))
          sequence))
```

- Individual functions remain understandable
- Still 2 reduces, an intersection and a mapcar
- Stratified design yields more useful functions

Rule of English Translation

To insure that you say what you mean:

1. Start with an English description of algorithm
2. Write the code from the description
3. Translate the code back into English
4. Compare 3 to 1

Example:

1. “Given a list of monsters, determine the number that are swarms.”

2.
```
(defun count-swarm (monster-list)
  (apply '+
         (mapcar
          #'(lambda (monster)
              (if (equal (object-type
                           (get-object monster))
                          'swarm)
                  1
                  0))
          monster-list)))
```

3. “Take the list of monsters and produce a 1 for a monster whose type is swarm, and a 0 for the others. Then add up the list of numbers.”

Rule of English Translation

Better:

1. “Given a list of monsters, determine the number that are swarms.”

2.
```
(defun count-swarms (monster-names)
  "Count the swarms in a list of monster names."
  (count-if #'swarm-p monster-names :key #'get-object))
```
   or
```
(count 'swarm monster-names :key #'get-object-type)
```
   or
```
(loop for name in monster-names
      count (swarm-p (get-object monster)))
```

3. “Given a list of monster names, count the number that are swarms.”

Use Library Functions

Libraries may have access to low-level efficiency hacks, and are often fine-tuned.

BUT they may be too general, hence inefficient.

Write a specific version when efficiency is a problem.

Good: *specific, concise*

```
(defun find-character (char string)
  "See if the character appears in the string."
  (find char string))
```

Good: *efficient*

```
(defun find-character (char string)
  "See if the character appears in the string."
  (declare (character char) (simple-string string))
  (loop for ch across string
        when (eql ch char) return ch))
```

Use Library Functions

Given `build1`, which maps n to a list of n `x`'s:
`(build1 4)` $\Rightarrow$ `(x x x x)`

Task: *Define* `build-it` *so that:*
`(build-it '(4 0 3))` $\Rightarrow$ `((x x x x) () (x x x))`

Incredibly Bad:

```
(defun round3 (x)
  (let ((result '()))
    (dotimes (n (length x) result)
      (setq result (cons (car (nthcdr n x)) result)))))

(defun build-it (arg-list)
  (let ((result '()))
    (dolist (a (round3 arg-list) result)
      (setq result (cons (build1 a) result)))))
```

Problems:

- `round3` is just another name for `reverse`
- `(car (nthcdr n x))` is `(nth n x)`
- `dolist` would be better than `dotimes` here
- `push` would be appropriate here
- `(mapcar #'build1 numbers)` does it all

Control Abstraction

Most algorithms can be characterized as:

- Searching (`some find find-if mismatch`)
- Sorting (`sort merge remove-duplicates`)
- Filtering (`remove remove-if mapcan`)
- Mapping (`map mapcar mapc`)
- Combining (`reduce mapcan`)
- Counting (`count count-if`)

These functions abstract common control patterns. Code that uses them is:

- Concise
- Self-documenting
- Easy to understand
- Often reusable
- Usually efficient
  (Better than a non-tail recursion)

Introducing your own control abstraction is an important part of stratified design.

Recursion vs. Iteration

Recursion is good for recursive data structures. Many people prefer to view a list as a sequence and use iteration over it, thus de-emphasizing the implementation detail that the list is split into a first and rest.

As an expressive style, tail recursion is often considered elegant. However, Common Lisp does not guarantee tail recursion elimination so it should not be used as a substitute for iteration in completely portable code. (In Scheme it is fine.)

The Common Lisp `do` macro can be thought of as syntactic sugar for tail recursion, where the initial values for variables are the argument values on the first function call, and the step values are argument values for subsequent function calls.

`do` provides a low level of abstraction, but versatile and has a simple, explicit execution model.

Recursion vs. Iteration (cont'd)

Bad: *(in Common Lisp)*

```
(defun any (lst)
  (cond ((null lst) nil)
        ((car lst) t)
        (t (any (cdr lst)))))
```

Better: *conventional, concise*

```
(defun any (list)
  "Return true if any member of list is true."
  (some #'not-null list))
```

or `(find-if-not #'null lst)`

or `(loop for x in list thereis x)`

or (explicit)

```
  (do ((list list (rest list)))
      ((null list) nil)
    (when (first list))
      (return t))))
```

Best: *efficient, most concise in this case*

Don't call `any` at all!

Use `(some` *p* `list)` instead of `(any (mapcar` *p* `list))`

LOOP

"Keep a `loop` to one topic—like a letter to your Senator." – *Judy Anderson*

The Common Lisp `loop` macro gives you the power to express idiomatic usages concisely. However it bears the burden that its syntax and semantics are often substantially more complex than its alternatives.

Whether or not to use the `loop` macro is an issue surrounded in controversy, and borders on a religious war. At the root of the conflict is the following somewhat paradoxical observation:

- `loop` appeals to naive programmers because it looks like English and seems to call for less knowledge of programming than its alternatives.
- `loop` is not English; its syntax and semantics have subtle intricacies that have been the source of many programming bugs. It is often best used by people who've taken the time to study and understand it—usually not naive programmers.

Use the unique features of loop (*e.g.*, parallel iteration of different kinds).

Simple Iteration

Bad: *verbose, control structure unclear*

```
(LOOP
  (SETQ *WORD* (POP *SENTENCE*))   ;get the next word
  (COND
   ;; if no more words then return instantiated CD form
   ;; which is stored in the variable *CONCEPT*
   ((NULL *WORD*)
    (RETURN (REMOVE-VARIABLES (VAR-VALUE '*CONCEPT*))))
   (T (FORMAT T "~%~%Processing ~A" *WORD*)
      (LOAD-DEF)        ; look up requests under
                        ; this word
      (RUN-STACK)))))   ; fire requests
```

- No need for global variables
- End test is misleading
- Not immediately clear what is done to each word

Good: *conventional, concise, explicit*

```
  (mapc #'process-word sentence)
  (remove-variables (var-value '*concept*))

(defun process-word (word)
  (format t "~2%Processing ~A" word)
  (load-def word)
  (run-stack))
```

Mapping

Bad: *verbose*

```
; (extract-id-list 'l_user-recs) ------------- [lambda]
; WHERE:   l_user-recs is a list of user records
; RETURNS: a list of all user id's in l_user-recs
; USES:    extract-id
; USED BY: process-users, sort-users

(defun extract-id-list (user-recs)
  (prog (id-list)
   loop
     (cond ((null user-recs)
            ;; id-list was constructed in reverse order
            ;; using cons, so it must be reversed now:
            (return (nreverse id-list))))
     (setq id-list (cons (extract-id (car user-recs))
                         id-list))
     (setq user-recs (cdr user-recs)) ;next user record
     (go loop)))
```

Good: *conventional, concise*

```
(defun extract-id-list (user-record-list)
  "Return the user ID's for a list of users."
  (mapcar #'extract-id user-record-list))
```

Counting

Bad: *verbose*

```
(defun size ()
  (prog (size idx)
     (setq size 0 idx 0)
   loop
     (cond ((< idx table-size)
            (setq size (+ size (length (aref table idx)))
                  idx (1+ idx))
            (go loop)))
     (return size)))
```

Good: *conventional, concise*

```
(defun table-count (table)    ; Formerly called SIZE
  "Count the number of keys in a hash-like table."
  (reduce #'+ table :key #'length))
```

Also, it couldn't hurt to add:

```
(deftype table ()
  "A table is a vector of buckets, where each bucket
  holds an alist of (key . values) pairs."
  '(vector cons))
```

Filtering

Bad: *verbose*

```
(defun remove-bad-pred-visited (l badpred closed)
  ;;; Returns a list of nodes in L that are not bad
  ;;; and are not in the CLOSED list.
  (cond ((null l) l)
        ((or (funcall badpred (car l))
             (member (car l) closed))
         (remove-bad-pred-visited
          (cdr l) badpred closed))
        (t (cons (car l)
                 (remove-bad-pred-visited
                  (cdr l) badpred closed)))))
```

Good: *conventional, concise*

```
(defun remove-bad-or-closed-nodes (nodes bad-node? closed)
  "Remove nodes that are bad or are on closed list"
  (remove-if #'(lambda (node)
                 (or (funcall bad-node? node)
                     (member node closed)))
             nodes))
```

Control Flow: Keep It Simple

Non-local control flow is hard to understand

Bad: *verbose, violates referential transparency*

```
(defun isa-test (x y n)
  (catch 'isa (isa-test1 x y n)))

(defun isa-test1 (x y n)
  (cond ((eq x y) t)
        ((member y (get x 'isa)) (throw 'isa t))
        ((zerop n) nil)
        (t (any (mapcar
                 #'(lambda (xx)
                     (isa-test xx y (1- n)) )
                 (get x 'isa) ))) ) )
```

Problems:

- `catch/throw` is gratuitous
- `member` test may or may not be helping
- `mapcar` generates garbage
- `any` tests too late;
  `throw` tries to fix this
  result is that `any` never gets called!

Keep It Simple

Some recommendations for use of `catch` and `throw`:

- Use `catch` and `throw` as sub-primitives when implementing more abstract control structures as macros, but do not use them in normal code.
- Sometimes when you establish a catch, programs may need to test for its presence. In that case, restarts may be more appropriate.

Keep It Simple

Good:

```
(defun isa-test (sub super max-depth)
  "Test if SUB is linked to SUPER by a chain of ISA
  links shorter than max-depth."
  (and (>= max-depth 0)
       (or (eq sub super)
           (some #'(lambda (parent)
                     (isa-test parent super
                               (- max-depth 1)))
                 (get sub 'isa)))))
```

Also good: *uses tools*

```
(defun isa-test (sub super max-depth)
  (depth-first-search :start sub :goal (is super)
                      :successors #'get-isa
                      :max-depth max-depth))
```

“Write clearly—don't be too clever.”
– *Kernighan & Plauger*

Be Aware:

Does “improving” something change the semantics? Does that matter?

Avoid Complicated Lambda Expressions

When a higher-order function would need a complicated lambda expression, consider alternatives:

- `dolist` or `loop`
- generate an intermediate (garbage) sequence
- Series
- Macros or read macros
- local function
  - Specific: makes it clear where function is used
  - Doesn't clutter up global name space
  - Local variables needn't be arguments
  - BUT: some debugging tools won't work

Avoid Complicated Lambda Expressions

Find the sum of the squares of the odd numbers in a list of integers:

All Good:

```
(reduce #'+ numbers
        :key #'(lambda (x) (if (oddp x) (* x x) 0)))

(flet ((square-odd (x) (if (oddp x) (* x x) 0)))
  (reduce #'+ numbers :key #'square-odd))

(loop for x in list
      when (oddp x) sum (* x x))

(collect-sum (choose-if #'oddp numbers))
```

Also consider: *(may be appropriate sometimes)*

```
;; Introduce read macro:
(reduce #'+ numbers :key #L(if (oddp _) (* _ _) 0))

;; Generate intermediate garbage:
(reduce #'+ (remove #'evenp (mapcar #'square numbers)))
```

Functional vs. Imperative Style

It has been argued that imperative style programs are harder to reason about. Here is a bug that stems from an imperative approach:

Task: Write a version of the built-in function `find`.

Bad: *incorrect*

```
(defun i-find (item seq &key (test #'eql) (test-not nil)
               (start 0 s-flag) (end nil)
               (key #'identity) (from-end nil))
  (if s-flag (setq seq (subseq seq start)))
  (if end (setq seq (subseq seq 0 end)))
  ...)
```

Problems:

- Taking subsequences generates garbage
- No appreciation of list/vector differences
- Error if both start and end are given
  Error stems from the update to `seq`

Example: Simplification

Task: a simplifier for logical expressions:

```
(simp '(and (and a b) (and (or c (or d e)) f)))
⇒ (AND A B (OR C D E) F)
```

Not bad, but not perfect:

```
(defun simp (pred)
  (cond ((atom pred) pred)
        ((eq (car pred) 'and)
         (cons 'and (simp-aux 'and (cdr pred))))
        ((eq (car pred) 'or)
         (cons 'or (simp-aux 'or (cdr pred))))
        (t pred)))

(defun simp-aux (op preds)
  (cond ((null preds) nil)
        ((and (listp (car preds))
              (eq (caar preds) op))
         (append (simp-aux op (cdar preds))
                 (simp-aux op (cdr preds))))
        (t (cons (simp (car preds))
                 (simp-aux op (cdr preds))))))
```

A Program to Simplify Expressions

Problems:

- No meaningful name for simp-aux
- No reusable parts
- No data accessors
- `(and)`, `(and a)` not simplified

Better: *usable tools*

```
(defun simp-bool (exp)
  "Simplify a boolean (and/or) expression."
  (cond ((atom exp) exp)
        ((member (op exp) '(and or))
         (maybe-add (op exp)
                    (collect-args
                     (op exp)
                     (mapcar #'simp-bool (args exp)))))
        (t exp)))

(defun collect-args (op args)
  "Return the list of args, splicing in args
  that have the given operator, op.  Useful for
  simplifying exps with associate operators."
  (loop for arg in args
        when (starts-with arg op)
        nconc (collect-args op (args arg))
        else collect arg))
```

Build Reusable Tools

```
(defun starts-with (list element)
  "Is this a list that starts with the given element?"
  (and (consp list)
       (eql (first list) element)))

(defun maybe-add (op args &optional
                     (default (get-identity op)))
  "If 1 arg, return it; if 0, return the default.
  If there is more than 1 arg, cons op on them.
  Example: (maybe-add 'progn '((f x))) ==> (f x)
  Example: (maybe-add '* '(3 4)) ==> (* 3 4).
  Example: (maybe-add '+ '()) ==> 0,
  assuming 0 is defined as the identity for +."
  (cond ((null args) default)
        ((length=1 args) (first args))
        (t (cons op args))))

(deftable identity
  :init '((+ 0) (* 1) (and t) (or nil) (progn nil)))
```

A Language for Simplifying

Task: A Simplifier for all Expressions:

```
(simplify '(* 1 (+ x (- y y))))   ==>  x
(simplify '(if (= 0 1) (f x)))    ==>  nil
(simplify '(and a (and (and) b))) ==> (and a b)
```

Syntactic abstraction defines a new language that is appropriate to the problem.

This is a problem-oriented (as opposed to code-oriented) approach.

Define a language for simplification rules, then write some:

```
(define-simplifier exp-simplifier
  ((+ x 0) ==> x)
  ((+ 0 x) ==> x)
  ((- x 0) ==> x)
  ((- x x) ==> 0)
  ((if t x y) ==> x)
  ((if nil x y) ==> y)
  ((if x y y) ==> y)
  ((and) ==> t)
  ((and x) ==> x)
  ((and x x) ==> x)
  ((and t x) ==> x)
  ...)
```

Design Your Language Carefully

“The ability to change notations empowers human beings.” – *Scott Kim*

Bad: *verbose, brittle*

```
(setq times0-rule '(
  simplify
  (* (? e1) 0)
  0
  times0-rule
  ) )

(setq rules (list times0-rule ...))
```

- Insufficient abstraction
- Requires naming `times0-rule` three times
- Introduces unneeded global variables
- Unconventional indentation

Sometimes it is useful to name rules:

```
(defrule times0-rule
  (* ?x 0) ==> 0)
```

(Although I wouldn’t recommend it in this case.)

An Interpreter for Simplifying

Now write an interpreter (or a compiler):

```
(defun simplify (exp)
  "Simplify expression by first simplifying components."
  (if (atom exp)
      exp
      (simplify-exp (mapcar #'simplify exp))))

(defun-memo simplify-exp (exp)
  "Simplify expression using a rule, or math."
  ;; The expression is non-atomic.
  (rule-based-translator exp *simplification-rules*
    :rule-pattern #'first
    :rule-response #'third
    :action #'simplify
    :otherwise #'eval-exp))
```

This solution is good because:

- Simplification rules are easy to write
- Control flow is abstracted away (mostly)
- It is easy to verify the rules are correct
- The program can quickly be up and running.
  If the approach is sufficient, we're done.
  If the approach is insufficient, we've saved time.
  If it is just slow, we can improve the tools,
  and other uses of the tools will benefit too.

An Interpreter for Translating

“Success comes from doing the same thing over and over again; each time you learn a little bit and you do a little better the next time.” – *Jonathan Sachs*

Abstract out the rule-based translator:

```
(defun rule-based-translator
    (input rules &key (matcher #'pat-match)
           (rule-pattern #'first) (rule-response #'rest)
           (action #identity) (sub #'sublis)
           (otherwise #'identity))
  "Find the first rule that matches input, and apply the
  action to the result of substituting the match result
  into the rule's response. If no rule matches, apply
  otherwise to the input."
  (loop for rule in rules
        for result = (funcall matcher
                       (funcall rule-pattern rule) input)
        when (not (eq result fail))
        do (RETURN (funcall action
                     (funcall sub result
                       (funcall rule-response rule))))
        finally (RETURN (funcall otherwise input))))
```

If this implementation is too slow, we can index better or compile.

Sometimes, reuse is at an informal level: seeing how the general tool is built allows a programmer to construct a custom tool with cut and paste.

Saving duplicate work: defun-memo

Less extreme than defining a whole new language is to augment the Lisp language with new macros.

`defun-memo` makes a function remember all computations it has made. It does this by maintaining a hash table of input/output pairs. If the first argument is just the function name, 1 of 2 things happen: [1] If there is exactly 1 arg and it is not a `&rest` arg, it makes a eql table on that arg. [2] Otherwise, it makes an equal table on the whole arglist.

You can also replace fn-name with (name :test ... :size ... :key-exp ...). This makes a table with given test and size, indexed by key-exp. The hash table can be cleared with the clear-memo function.

Examples:

```
(defun-memo f (x)               ;; eql table keyed on x
  (complex-computation x))

(defun-memo (f :test #'eq) (x) ;; eq table keyed on x
  (complex-computation x))

(defun-memo g (x y z)           ;; equal table
  (another-computation x y z)) ;; keyed on on (x y . z)

(defun-memo (h :key-exp x) (x &optional debug?)
                                ;; eql table keyed on x
  ...)
```

Saving Duplicate Work: defun-memo

```
(defmacro defun-memo (fn-name-and-options (&rest args)
                                          &body body)
  ;; Documentation string on previous page
  (let ((vars (arglist-vars args)))
    (flet ((gen-body (fn-name &key (test '#'equal)
                                size key-exp)
             `(eval-when (load eval compile)
               (setf (get ',fn-name 'memoize-table)
                (make-hash-table :test ,test
                 ,@(when size `(:size ,size))))
               (defun ,fn-name ,args
                 (gethash-or-set-default
                  ,key-exp
                  (get ',fn-name 'memoize-table)
                  (progn ,@body))))))
      ;; Body of the macro:
      (cond ((consp fn-name-and-options)
             ;; Use user-supplied keywords, if any
             (apply #'gen-body fn-name-and-options))
            ((and (= (length vars) 1)
                  (not (member '&rest args)))
             ;; Use eql table if it seems reasonable
             (gen-body fn-name-and-options :test '#'eql
                       :key-exp (first vars)))
            (t ; Otherwise use equal table on all args
             (gen-body fn-name-and-options :test '#'equal
                       :key-exp `(list* ,@vars)))))))
```

More Macros

```
(defmacro with-gensyms (symbols body)
  "Replace the given symbols with gensym-ed versions,
  everywhere in body.  Useful for macros."
  ;; Does this everywhere, not just for "variables"
  (sublis (mapcar #'(lambda (sym)
                      (cons sym (gensym (string sym))))
                  symbols)
          body))

(defmacro gethash-or-set-default (key table default)
  "Get the value from table, or set it to the default.
  Doesn't evaluate the default unless needed."
  (with-gensyms (keyvar tabvar val found-p)
    `(let ((keyvar ,key)
           (tabvar ,table))
      (multiple-value-bind (val found-p)
          (gethash keyvar tabvar)
        (if found-p
            val
            (setf (gethash keyvar tabvar)
                  ,default))))))
```

Use Macros Appropriately

(See tutorial by Allan Wechsler)

The design of macros:

- Decide if a macro is really necessary
- Pick a clear, consistent syntax for the macro
- Figure out the right expansion
- Use **`defmacro`** and ‘ to implement the mapping
- In most cases, also provide a functional interface (useful, sometimes easier to alter and continue)

Things to think about:

- Don’t use a macro where a function would suffice
- Make sure nothing is done at expansion time (mostly)
- Evaluate args left-to-right, once each (if at all)
- Don’t clash with user names (with-gensyms)

Problems with Macros

Bad: *should be an inline function*

```
(defmacro name-part-of (rule)
  `(car ,rule))
```

Bad: *should be a function*

```
(defmacro defpredfun (name evaluation-function)
  `(push (make-predfun :name ,name
           :evaluation-function ,evaluation-function)
         *predicate-functions*))
```

Bad: *works at expansion time*

```
(defmacro defclass (name &rest def)
  (setf (get name 'class) def)
  ...
  (list 'quote name))
```

Problems with Macros

Bad: *Macros should not eval args*

```
(defmacro add-person (name mother father sex
                              unevaluated-age)
  (let ((age (eval unevaluated-age)))
    (list (if (< age 16) ... ...) ...)))

(add-person bob joanne jim male (compute-age 1953))
```

What if you compiled this call now and loaded it in a few years?

Better: *Let the compiler constant-fold*

```
(declaim (inline compute-age))

(defmacro add-person (name mother father sex age)
  `(funcall (if (< ,age 16) ... ...) ...)))
```

Very Bad: *(what if increment is n?)*

```
(defmacro for ((variable start end &optional increment)
               &body body)
  (if (not (numberp increment)) (setf increment 1))
  ...)

(for (i 1 10) ...)
```

Macros for Control Structures

Good: *fills a hole in orthogonality of CL*

```
(defmacro dovector ((var vector &key (start 0) end)
                    &body body)
  "Do body with var bound to each element of vector.
  You can specify a subrange of the vector."
  `(block nil
    (map-vector #'(lambda (,var) ,@body)
                ,vector :start start :end end)))

(defun map-vector (fn vector &key (start 0) end)
  "Call fn on each element of vector within a range."
  (loop for i from start below (or end (length vector))
        do (funcall fn (aref vector-var index))))
```

- Iterates over a common data type
- Follows established syntax (`dolist, dotimes`)
- Obeys declarations, returns
- Extends established syntax with keywords
- One bad point:
  No result as in `dolist`, `dotimes`

Helper Functions For Macros

Most macros should expand into a call to a function.

The real work of the macro `dovector` is done by a function, `map-vector` because:

- It's easier to patch
- It's separately callable (useful for program)
- The resulting code is smaller
- If prefered, the helper can be made inline
  (Often good to avoid consing closures)

```
(dovector (x vect) (print x))
```

macro-expands to:

```
(block nil
 (map-vector #'(lambda (x) (print x)) vect
             :start 0 :end nil))
```

which inline expands to (roughly):

```
(loop for i from 0 below (length vect)
        do (print (aref vect i)))
```

Setf Methods

As in macros, we need to be sure to evaluate each form exactly once, in left-to-right order.

Make sure macro expansions (`macroexpand`, `get-setf-method`) are done in the right environment.

```
(defmacro deletef (item sequence &rest keys
                        &environment environment)
  "Destructively delete item from sequence."
  (multiple-value-bind (temps vals stores store-form
                              access-form)
      (get-setf-method sequence environment)
    (assert (= (length stores) 1))
    (let ((item-var (gensym "ITEM")))
      `(let* ((,item-var ,item)
              ,@(mapcar #'list temps vals)
              (,(first stores)
               (delete ,item-var ,access-form ,@keys)))
         ,store-form))))
```

Programming in the Large

Be aware of stages of software development:

- Gathering requirements
- Architecture
- Component Design
- Implementation
- Debugging
- Tuning

These can overlap. The point of exploratory programming is to minimize component design time, getting quickly to implementation in order to decide if the architecture and requirements are right.

Know how to put together a large program:

- Using packages
- Using `defsystem`
- Separating source code into files
- Documentation in the large
- Portability
- Error handling
- Interfacing with non-Lisp programs

Separating Source Code into Files

The following factors affect how code is decomposed into files

- Language-imposed dependencies
  macros, inline functions, CLOS classes before use
- Stratified design
  isolate reusable components
- Functional decomposition
  group related components
- Compatibility with tools
  chose good size files for editor, `compile-file`
- Separate OS/machine/vendor-specific implementations

Using Comments Effectively

Use comments to/for:

- **Explain philosophy.** Don't just document details; also document philosophy, motivation, and metaphors that provide a framework for understanding the overall structure of the code.
- **Offer examples.** Sometimes an example is worth a pile of documentation.
- **Have conversations with other developers!** In a collaborative project, you can sometimes ask a question just by putting it in the source. You may come back to find it answered. Leave the question and the answer for others who might later wonder, too.
- **Maintain your "to do" list.** Put a special marker on comments that you want to return to later: `???` or `!!!`; maybe use `!!!!` for higher priority. Some projects keep to do lists and change logs in files that are separate from the source code.

```
(defun factorial (n)
  ;; !!! What about negative numbers? --Joe 03-Aug-93
  ;; !!! And what about non-numbers?? -Bill 08-Aug-93
  (if (= n 0) 1
      (* n (factorial (- n 1)))))
```

Documentation: Say What You Mean

Q: Do you ever use comments when you write code?

"Rarely, except at the beginning of procedures, and then I only comment on the data structure. I don't comments on the code itself because I feel that properly written code is very self-documented." – *Gary Kildall*

"I figure there are two types of comments: one is explaining the obvious, and those are worse than worthless, the other kind is when you explain really involved, convoluted code. Well. I always try to avoid convoluted code. I try to program really strong, clear, clean code, even if it makes an extra five lines. I am almost of the opinion that the more comments you need, the worse your program is and something is wrong with it." – *Wayne Ratliff*

"Don't comment bad code—rewrite it." – *Kernighan & Plauger*

- Describe the purpose and structure of system
- Describe each file
- Describe each package
- Documentation strings for all functions
- Consider automatic tools (`manual`)
- Make code, not comments

Documentation: Over-commenting

These 32-lines *must* document a major system:

```
; ===================================================================
;
; describe
; --------
;
;  arguments     : snepsul-exp - <snepsul-exp>
;
;  returns       : <node set>
;
;  description   : This calls "sneval" to evaluate "snepsul-exp" to
;                  get the desired <node set>.
;                  It prints the description of each <node> in the
;                  <node set> that has not yet been described during
;                  the process; the description includes the
;                  description of all <node>s dominated by the <node>.
;                  It returns the <node set>.
;
;  implementation: Stores the <node>s which have already been described
;                  in "describe-nodes".
;                  Before tracing the description of a <node>, it
;                  checks whether the <node> was already been described
;                  to avoid describing the same <node> repeatedly.
;                  The variable "describe-nodes" is updated by "des1".
;
;  side-effects  : Prints the <node>'s descriptions.
;
;                                      written:  CCC 07/28/83
;                                      modified: CCC 09/26/83
;                                                ejm 10/10/83
;                                                njm 09/28/88
;                                                njm  4/27/89
```

Documentation: Over-commenting

```
(defmacro describe (&rest snepsul-exp)
  `(let* ((crntct (processcontextdescr ',snepsul-exp))
          (ns (in-context.ns (nseval (getsndescr
                                      ',snepsul-exp))
                             crntct))
          (described-nodes (new.ns))
          (full nil))
     (declare (special crntct described-nodes full))
     (terpri)
     (mapc #'(lambda (n)
               (if (not (ismemb.ns n described-nodes))
                   (PP-nodetree (des1 n))))
           ns)
     (terpri)
     (values ns crntct)))
```

Problems:

- Documentation too long; lose big picture
- Documentation is wrong: describe(d)-nodes.
- Documentation is ineffective: no doc string
- Documentation is redundant (arglist)
- Bad idea to shadow Lisp's describe function
- Need function that is separate from macro
- Abbreviations are obscure

Documentation: Commenting

Better:

This doesn't handle crntct (whatever that is)

```
(defmacro desc (&rest snepsul-exp)
  "Describe the node referred to by this expression.
  This macro is intended as an interactive debugging tool;
  use the function describe-node-set from a program."
  `(describe-node-set (exp->node-set ',snepsul-exp)))

(defun describe-node-set (node-set)
  "Print all the nodes in this node set."
  ;; Accumulate described-nodes to weed out duplicates.
  (let ((described-nodes (new-node-set)))
    (terpri)
    (dolist (node node-set)
      (unless (is-member-node-set node described-nodes)
        ;; des1 adds nodes to described-nodes
        (pp-nodetree (des1 node described-nodes))))
    (terpri)
    node-set))
```

Portability

Make your program run well in the environment(s) you use.

But be aware that you or someone else may want to use it in another environment someday.

- Use **#+***feature* and **#-***feature*
- Isolate implementation-dependent parts.
- Maintain one source and multiple binaries
- Evolve towards dpANS CL
  Implement missing features if needed
- Be aware of vendor-specific extensions

Foreign Function Interface

Large programs often have to interface with other programs written in other languages. Unfortunately, there is no standard for this.

- Learn your vendor's foreign interface
- Try to minimize exchange of data
- Beware of areas that cause problems:
  Memory management
  Signal handling

Mean what you say

- Don't mislead the reader
  Anticipate reader's misunderstandings
- Use the right level of specificity
- Be careful with declarations
  Incorrect declarations can break code
- One-to-one correspondence

Bad declaration: *only made-up example*

```
(defun lookup (name)
  (declare (type string name))
  (if (null name)
      nil
      (or (gethash name *symbol-table*)
          (make-symbol-entry name))))
```

Should be `(declare (type (or string null) name))`

Naming Conventions: Be Consistent

Be consistent in names:

- Be consistent with capitalization
  most prefer `like-this`, not `LikeThis`
- `*special-variable*`
- `+constant+` (or some convention)
- Dylan uses `<class>`
- Consider `structure.slot`
- `-p` or ?; ! or `n`; `->` or `-to-`
- verb-object: `delete-file`
  object-attribute: `integer-length`
  compare `name-file` and `file-name`
  don't use object-verb or attribute-object!
- Order arguments consistently
- Distinguish internal and external functions
  Don't mix `&optional` and `&key`; use carefully
  1 or 2 `&optional` args (Dylan 0)
  Use keywords consistently (`key, test, end`)

Naming Conventions: Choose Names Wisely

Choose Names wisely:

- Minimize abbreviations

  Most words have many possible abbreviations but only one correct spelling. Spell out names so they are easier to read, remember, and find.

  Some possible exceptions: char, demo, intro, and paren. These words are becoming almost like real words in English. A good test (for native English speakers) is: Would you say the word aloud in conversation? Our earlier example with `crntct` and `processcontextdescr` wouldn't pass this test.

- Don't shadow a local variable with another.
- Clearly show variables that are updated.
- Avoid ambiguous names; Use `previous` or `final` instead of `last`.

Notational Tricks: Parens in Column 0

Most text editors treat a left paren in column 0 as the start of a top-level expression. A paren inside a string in column 0 may confuse the editor unless you provide a backslash:

```
(defun factorial (n)
  "Compute the factorial of an integer.
\(don't worry about non-integer args)."
  (if (= n 0) 1
      (* n (factorial (- n 1)))))
```

Many text editors will treat a "`(def`" in column 0 as a definition, but not a "`(def`" in other columns. So you may need to do this:

```
(progn
(defun foo ...)
(defun bar ...)
)
```

Multi-Line Strings

In case of a multi-line string as a literal constant, such as:

```
(defun find-subject-line (message-header-string)
  (search "
Subject:" message-header-string))
```

consider instead using read-time evaluation and a call to `format`:

```
(defun find-subject-line (message-header-string)
  (search #.(format nil "~%Subject:") message-header-string
```

Where the same string is used many times, consider using a global variable or named constant:

```
(defparameter *subject-marker* (format nil "~%Subject:"))

(defun find-subject-line (message-header-string)
  (search *subject-marker* message-header-string))
```

Multi-Line Strings (cont'd)

For long format strings, you can indent the continuation lines with `<Return>` or `@<Return>`. The following two forms do the same thing:

```
(format t "~&This is a long string.~@
             This is more of that string.")
This is a long string.
This is more of that string.
```

```
(format t "~&This is a long string.~
           ~%This is more of that string.")
This is a long string.
This is more of that string.
```

The latter syntax permits you to indent a fixed amount easily:

```
(format t "~&This is a long string.~
           ~% This is more of that string, indented by one.'
This is a long string.
 This is more of that string, indented by one.
```

Notational Tricks: Multi-Line Comments

Avoid using **#|** and **|#** in strings, since it will confuse any later attempt to comment out such a string. Again, a backslash helps:

Good:

```
(defun begin-comment () (write-string "#\|"))
(defun end-comment () (write-string "|\#"))
```

This means that you can later comment out sections containing these strings without editing the strings themselves.

If your editor provides support (comment-region and uncomment-region commands) it is better to use explicit ;; comments. That way the reader will never get confused about which sections have been commented out.

Some Red Flags

The following situations are "red flags." They are often symptoms of problems—even though technically most of them do happen in completely legitimate situations as well. If you see one of these red flags, you do not automatically have a problem in your code, but you should still proceed cautiously:

- Any use of `eval`
- Any use of `gentemp` *
- Any use of `append`
- The absence of an `&environment` parameter in a macro that uses `setf` or calls `macroexpand`.
- Writing a condition handler for type `error` (including use of `ignore-errors`).
- Any use of the `c...r` functions except `caar`, `cad...r`, (where the "`...`" is all `d`'s).

* No known good uses.

Avoid Common Mistakes

Good style involves avoiding mistakes.

- Always prompt for input
  (Or user won't know what's happening)
- Understand `defvar` and `defparameter`
- Understand `flet` and `labels`
- Understand multiple values
- Understand macros (shown above)
- Recompile after changing macros or
  inline functions
- Use `#'(lambda ...)`, not `'(lambda ...)`
- Remember `#'f` is just `(function f)`
- Use `:test #'equal` as needed
- Make sure declarations are effective
- Have a policy for destructive functions

Destructive Functions

Have a policy for destructive functions:

- Most programs use destructive updates when they can prove the arguments are not needed elsewhere (as when a function nconc's partial results).
- Otherwise, assume that arguments cannot be altered
- Assume that results will not be altered
- Major interfaces often make copies of results they pass out, just to be safe.
- Note that generation scavenging GC can be slowed down by destructive updates.

Minor Mistakes

Bad:

```
(defun combine-indep-lambdas (arc-exp)
  (apply #'*
         (mapcar #'eval-arc-exp (cdr arc-exp))))
```

- apply may exceed `call-arguments-limit`
- mapcar generates garbage
- cdr violates data abstraction

Good:

```
(reduce #'* (in-arcs arc-exp) :key #'eval-arc-exp)
```

Learn to use accumulators:

```
(defun product (numbers &optional (key #'identity)
                (accum 1))
  "Like (reduce #'* numbers), but bails out early
  when a zero is found."
  (if (null numbers)
      accum
      (let ((term (funcall key (first numbers))))
        (if (= term 0)
            0
            (product (rest numbers) key (* accum term))))))
```

Consider Series:

```
(collect-fn 'number (constantly 1) #'* numbers)
```

Multi-Tasking and Multi-Processing

Multi-Tasking (Time Slicing)

It is reasonable to spend time structuring your code to work well in the face of multi-tasking. Many commercial Lisp implementations have this even though there is not yet a portable standard. It fits in well with existing language semantics.

- Watch out for global state like `setq` and property lists.
- Synchronize processes with `without-interrupts`, `without-aborts`, `without-preemption`, etc. Consult implementation-specific documentation for the set of available operators and learn how they differ.

Multi-Processing (True Parallelism)

Think about true parallelism, but don't waste a lot of time structuring your programs to work well if things suddenly become parallelized. Making a sequential program into a parallel one is a non-trivial change that won't happen by accident (*e.g.,* due to some overnight change in Common Lisp's semantics). It will take a whole new language to support this; you'll have time to prepare.

Expect The Unexpected

Murphy's Law

"If something can go wrong, it will."

Don't omit checking for things because you're sure something will never happen unless you're very sure. ... And even then, don't omit them anyway. It is sufficiently commonplace to get errors from systems saying "This can't happen" that it's clear that people are not always as brilliant as they think.

Read Other People's Code

"You need to study other people's work. Their approaches to problem solving and the tools they use give you a fresh way to look at your own work." – *Gary Kildall*

"I've learned a lot from looking at other people's programs." – *Jonathan Sachs*

"I still think that one of the finest tests of programming ability is to hand the programmer about 30 pages of code and see how quickly he can read through and understand it." – *Bill Gates*

"The best way to prepare [to be a programmer] is to write programs, and to study great programs that other people have written. In my case, I went to the garbage cans at the Computer Science Center and I fished out listings of their operating system." – *Bill Gates*

"You've got to be willing to read other people's code, then write your own, then have other people review your code." – *Bill Gates*

- Lisp Machine Operating System
- Internet FTP sites (comp.lang.lisp FAQ)
- CMU CL Compiler and Utilities
- Macintosh Common Lisp examples

Example: deftable

Task: Make it easy to define and use tables.

- Like defstruct
- Should be fast: inline functions
- Should handle one or multiple tables
- CLOS?
- Operations? Arguments and return value(s)?
- Default values? Mutated or returned?

Separation between user and implementor code, with support for both.

- Way to define new table implementations
- Naming; packages?
- Documented limitations?
- Instrumentation?
- Automatic selection?

Lessons learned:

- Capture common abstractions: tables, others?
- Complex macros can be designed with a little care
- Consider the possibility of extension

Prototype

Lisp allows you to develop prototypes easily.

“Plan to throw one away; you will, anyhow.” – *Fred Brooks*

“I think a lot before I do anything, and once I do something, I’m not afraid to throw it away. It’s very important that a programmer be able to look back at a piece of code like a bad chapter in a book and scrap it without looking back.” – *John Warnock*

“Don’t bind early; don’t ever make decisions earlier than you have to. Stay an order of magnitude more general than you think you need, because you will end up needing it in the long term. Get something working very quickly and then be able to throw it away.” – *John Warnock*

“So I tend to write a few lines at a time and try it out, get it to work, then write a few more lines. I try to do the least amount of work per iteration to make real substantive change.” – *Wayne Ratliff*

“1-2-3 began with a working program, and it continued to be a working program throughout its development.” – *Jonathan Sachs*

Other Ideas

Learn to type. If you type less than 60 wpm, you're holding yourself back.

"Also, while you're working hard on a complicated program, it's important to exercise. The lack of physical exercise does most programmers in. It causes a loss of mental acuity." – *John Page*

Q: What does it take to become a great programmer?

"What does it take to be good at anything? What does it take to be a good writer? Someone who's good is a combination of two factors: an accidental mental correspondence to the needs of the disciplines, combined with a mental ability to not be stupid. That's a rare combination, but it's not at all mystical. A good programmer must enjoy programming and be interested in it, so he will try to learn more. A good programmer also needs an aesthetic sense, combined with a guilt complex, and a keen awareness of when to violate that aesthetic sense. The guilt complex forces him to work harder to improve the program and to bring it more in line with the aesthetic sense." – *Bob Frankston*

Recommended Books

Introductions to Common Lisp

- Robert Wilensky *Common LISPcraft*
- Deborah G. Tatar *A Programmer's Guide to Common Lisp*
- Rodney A. Brooks. *Programming in Common Lisp*

References and a Must-Have

- Guy L. Steele *Common Lisp: The Language, 2nd Edition*
- ANSI *Draft Proposed Common Lisp Standard*
- Harold Abelson and Gerald Jay Sussman, with Julie Sussman. *Structure and Interpretation of Computer Programs* (Scheme)

Recommended Books

More Advanced:

- Patrick H. Winston and Berthold K. P. Horn. *LISP*, 3rd edition.
- Wade L. Hennessey *Common Lisp*
- Sonya E. Keene *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*
- Eugene Charniak, Christopher K. Riesbeck, Drew V. McDermott and James R. Meehan. *Artificial Intelligence Programming*, 2nd edition.
- Peter Norvig. *Paradigms of AI Programming: Case Studies in Common Lisp*

Periodicals:

- LISP Pointers. (ACM SIGPLAN) Since 1987.
- LISP and Symbolic Computation. Since 1989.
- Proceedings of the biannual ACM Lisp and Functional Programming Conference. Since 1980.

Quotes

Quotes from *Programmers at Work*, Susan Lammers, Microsoft Press, 1989.

- Bob Frankston: Software Arts VisiCalc; Lotus
- Bill Gates: Altair BASIC; Microsoft
- Gary Kildall: Digital Research CP/M
- Scott Kim: Stanford, Xerox; Inversions
- Butler Lampson: Xerox Ethernet, Alto, Dorado, Star, Mesa; DEC
- John Page: HP; Software Publishing PFS:FILE
- Wayne Ratliff: NASA; Ashton-Tate dBASE II
- Jonathan Sachs: MIT; Lotus 1-2-3
- Charles Simonyi: Xerox Bravo; Microsoft Word, Excel
- John Warnock: NASA; Xerox; Adobe PostScript

Quotes

Other quotes:

- Harold Abelson: MIT; SICP; Logo
- Judy Anderson: Harlequin, Inc.
- Fred Brooks: IBM 360 architect, now at UNC
- Bear Bryant: Alabama football coach
- Brian Kernighan & P.J. Plauger: Bell Labs UNIX
- David McDonald: MIT, UMass natural language generation
- Guy Steele: Thinking Machines; Scheme; CLtL
- Gerald Sussman: MIT; SICP; Scheme
- Deborah Tatar: DEC; Xerox; author
- Niklaus Wirth: ETH Zurich; Pascal

Almost all Bad Code examples are taken from published books and articles (whose authors should know better, but will remain anonymous).