Common Lisp:
A Brief Overview

Maya Johnson, Haley Nolan,
and Adrian Velonis

Common Lisp Overview

- Common Lisp is a commonly used dialect of LISP
- 2nd oldest high-level programming language
- Both functional and object-oriented
- Based on mathematical notation, and the syntax relies heavily on parentheses

```
(print "Hello, world!")
```

Variables and Assignment Review

Local Variables

- Defined with let, redefined with setq/setf
- (let (<var> <expression>))
  Ex: (let (str "Hello, world!"))

Dynamic (Global) Variables

- Defined with defvar or defparameter
- (defvar <var> <expression>)
- (defparameter <var> <expression>)

Multiple Assignment with Local Variables

- Using let
```
(let ((<var1> <expression1>)
        (<var2> <expression2>)))
```
- Using multiple-value-bind
```
(multiple-value-bind <var-1 .. var-n>
     <expression> <optional body using var>)
```

Built-In Types and Operators Review

Integer Types
- bignum
- fixnum

Rational Type
- ratio

Floating Point Types
- short-float
- single-float
- double-float
- Long-float

Complex Numbers
- #C(1 1) or (complex (+ 1 2) 5)
- (realpart #C(7 9)) or (imagpart #C(7 9))

Boolean Type
- Value nil is false; all other values are true (usually t)

Comparison Operators for above types

=, /=, >, <, >=, <=, eql (checks type)

Character Type
- #\x: represents character ‘x’
- CHAR=, CHAR/=, CHAR<, CHAR>, etc.

String Type
- One-dimensional array of characters
- Compared using STRING=, STRING<, etc

Logical Operators
- and, or, not

Composite Data Types

Sequences: underlying structure of lists, vectors (1D arrays), and strings

Lists

- Special object nil which represents empty list
- Made up of cons cells, which are essentially nodes -> allows circular lists
  - Format is (cons car cdr)
  - Can be a two-element structure: (cons 1 2) -> (1.2)
  - Becomes a list if cdr of last cell is nil: (cons 1 (cons 2 nil)) -> (1 2)
- Literal list object: (list 1 2) -> (1 2) OR ‘(1 2) -> (1 2)
- Get length with (list-length <list>) -> returns length OR nil if circular
- Compare/use lists using set functions (e.g., union)

Arrays

- Dimensional collections of objects
- Create and modify using make-array and adjust-array
- (defparameter myarray (make-array '(2 2) :initial-element 1)) -> #2A((1 1) (1 1))
- 1-dimensional arrays are vectors

Hash Tables: map keys to values -> (setf (gethash 'one-entry *my-hash*) "one")

Alists: association lists, made up of cons cells -> (FOO . “foo”) (BAR . “bar”)

Plists: property lists, cons cells alternates keys and values -> FOO “foo” BAR “bar”

Composite Data Types

Structures

- Common Lisp's version of a struct
- Define using defstruct

```
(defstruct person
     name
     id
     birthday)
```

- Automatically populates some functions
  - Access functions to get inner variables (similar to "get" methods)
  - Type checking function person-p -> returns true if of type person
  - Constructor function make-person
  - Print function (similar to toString)
  - Copy function copy-person
- Can altered variables using (setf (name person1) "Bob")

Selection Review

Conditional Statements

```
-   (if <condition> <value if true> <value if false>)
    Ex: (if t 5 6) → 5

-   (cond (test then) (t else))
    Ex: (cond (t 5) (t 6)) → 5

-   (when <condition> <value>)
    Ex: (when t 5) → 5

-   (unless <condition> <value>)
    Ex: (unless t 5) → NIL
```

Iteration and Recursion Review

Iteration

- Built-in loop and do keywords

```
(loop for <var> in <list>
         do (<action>))
```

- Keyword dotimes

```
(dotimes (i 10)
     (print i))
```

- Macro iter

```
(iter (for <var> from <value1> to <value2>))
```

- No existing while loop, can define macro

```
(defmacro while (condition &body body)
     (loop while, condition do (progn ,@body)))
```

Recursion

- Recursion is an important feature of Common Lisp

Ex.:
```
(def factorial (x)
         (cond (= x 1 1)
         (t (* x (factorial (- x 1))))))
```

Subroutines

- Functions in Common Lisp are defined with the `defun` keyword
- Existing functions can be called with the terms `funcall` or `apply`
- Anonymous functions can be written using the `lambda` macro
- The syntax `#'` can be used to signify that the program is searching for a function name, rather than a value of the function

```
(defun add(x, y)
     (+ x y))
(defun hello-world()
     (format t "Hello, world!"))
(defun calladd()
     (funcall add(x y)))
(defun calladd2()
     (apply add(x y)))
```

```
(lambda (x)
     (= 0 (mod x 2)))
(funcall #'add(x y))
(funcall #'add '(1 2))
(funcall #'(lambda (x y) (+ x y)) 2 3)
```

Parameter Passing

- Uses call by sharing
  - All variables are references to object -> that reference is passed
  - Formal and actual parameters refer to the same object
- Allows for a fixed or variable number of parameters
  - User-defined functions have a fixed number by default
- Uses positional association by default, but allows named association

Variable Number of Parameters

- List any required arguments first, then `&rest` plus a name for the parameter list

2.https://ccrma.stanford.edu/courses/220b-winter-2005/topics/commonlisp/arguments.html

Format:

```
(defun func-name (required-parameters &rest args))
```

Example[2]:

```
(defun count-arguments (&rest args)
     (length args))

(count-arguments 1 2 3 4 5)
-> 5
(count-arguments)
-> 0
```

Named Parameters

- Use `&key` before any named parameters
  - They have a default value and are optional

2.https://ccrma.stanford.edu/courses/220b-winter-2005/topics/commonlisp/arguments.html

Format: `(defun func-name (&key (param-name1 default1) (param-name2 default2) …))`

Call: `(func-name :param-name1 value1 :param-name2 value2)`

Example[2]:

```
(defun poem (&key (rose-color 'red)
(violet-color 'blue))
   (list 'roses 'are rose-color 'and 'violets
'are violet-color))

(poem)
-> (roses are red and violets are blue)
(poem :violet-color 'violet :rose-color 'yellow)
-> (roses are yellow and violets are violet)
```

Data Abstraction Overview

- Common Lisp supports object-oriented programming

- It is class based (every object is an instance of a class)
  - Every class is a subclass of the root class T (done implicitly)

- Users can define new classes
  - Methods are associated with these classes through generic functions (encapsulation)

- Supports multiple inheritance

Defining Classes

Format of a class:

```
(defclass <class-name> (list of super
classes)
  ((slot-1
     :slot-option slot-argument)
   (slot-2, etc))
 )
```

These are all valid definitions of the person class:

```
(defclass person ()
  ((name
    :initarg :name
    :accessor name)
   (lisper
    :initform nil
    :accessor lisper)))

(defclass person ()
  (name lisper))

(defclass person () )
```

Defining Classes

- Instances of classes are created with make-instance
  - But good practice to define a constructor for it

```
(defvar p1 (make-instance 'person :name
"me"))

(defun make-person (name &key lisper)
  (make-instance 'person :name name :lisper
          lisper))
```

Accessing Variables in Classes

- Variables in classes are accessible at any point outside the class
  - Accessed using “slot-value”

Format: (slot-value <object> <slot-name>)

```
(defvar p1 (make-instance 'person
    :name "Bryn"))

(slot-value p1 'name)
-> "Bryn"

(setf (slot-value p1 'lisper "yes"))

(slot-value p1 'lisper)
-> "yes"
```

Generic Functions

- Core of Common Lisp's object-oriented-ness
- How classes are associated with behaviors
  - The generic function takes the class it's associated with as a parameter
  - Its subclasses inherit this function (like how circle inherited shape's function)

```
(defclass shape () )

(defgeneric calc-area (shape)
  (:documentation "calculate the area
of the shape"))


(defclass circle (shape)
  (radius))

(defmethod calc-area ((shape circle))
  (* pi (* radius radius)))
```

Standard Method Combination

- Four types of methods: primary, before, after, and around
    - All functions shown previously have been primary functions
- Before methods get called *before* the primary method, after methods *after*, and around methods when relevant and called by call-next-method
- The type is declared with a method qualifier (if none, primary is assumed)

```
(defmethod method-name :before (...) ...)
(defmethod method-name :after  (...) ...)
(defmethod method-name :around (...) ...)
```

Before and After Methods

```
; Define a primary method
(defmethod combo1 ((x number)) (print 'primary))

; Define before methods
(defmethod combo1 :before ((x integer))
     (print 'before-integer))
(defmethod combo1 :before ((x rational))
     (print 'before-rational))

; Define after methods
(defmethod combo1 :after ((x integer))
     (print 'after-integer))

(defmethod combo1 :after ((x rational))
     (print 'after-rational))
```

```
(combo1 17)
-> BEFORE-INTEGER
-> BEFORE-RATIONAL
-> PRIMARY
-> AFTER-RATIONAL
-> AFTER-INTEGER
```

```
(combo1 4/5)
-> BEFORE-RATIONAL
-> PRIMARY
-> AFTER-RATIONAL
```

Example from 12. https://dept-info.labri.fr/~strandh/Teaching/MTP/Common/David-Lamkins/chapter14.html

Around Methods and call-next-method

```
; Define a primary method
(defmethod combo2 ((x number)) (print 'primary))

; Define a before method and after method
(defmethod combo2 :before ((x integer)) (print 'before-integer))
(defmethod combo2 :after ((x integer)) (print 'after-integer))

; Define around methods
(defmethod combo2 :around ((x float))
    (print 'around-float-before-call-next-method)
    (let ((result (call-next-method (float (truncate x)))))
      (print 'around-float-after-call-next-method)
      result))
(defmethod combo2 :around ((x number))
    (print 'around-number-before-call-next-method)
    (print (call-next-method))
    (print 'around-number-after-call-next-method))
```

Example from 12. https://dept-info.labri.fr/~strandh/Teaching/MTP/Common/David-Lamkins/chapter14.html

```
(combo2 17)
-> AROUND-NUMBER-BEFORE-CALL-NEXT-METHOD
-> BEFORE-INTEGER
-> PRIMARY
-> AFTER-INTEGER
-> AROUND-NUMBER-AFTER-CALL-NEXT-METHOD
```

```
(combo2 82.3)
-> AROUND-FLOAT-BEFORE-CALL-NEXT-METHOD
-> AROUND-NUMBER-BEFORE-CALL-NEXT-METHOD
-> PRIMARY
-> AROUND-NUMBER-AFTER-CALL-NEXT-METHOD
-> AROUND-FLOAT-AFTER-CALL-NEXT-METHOD
```

Exception Handling

Common Lisp uses conditions to represent errors/exceptions or places in a program where there are branches in logic

Creating Conditions

- Built-in conditions
- User-defined conditions: define using define-condition and initialize using make-condition

Throwing Conditions

- Can throw using error or warn
- Depends on whether opening debugger
- Also has simple form

```
(define-condition my-division-by-zero (error)
  ((dividend :initarg :dividend
             :initform nil
             :reader dividend))
  (:report (lambda (condition stream)
     (format stream "You were going to divide ~a by
             zero.~&" (dividend condition)))))

(make-condition 'my-division-by-zero :dividend 3)
```

```
(error 'my-division-by-zero :dividend 3)
;; Debugger:
;;
;; You were going to divide 3 by zero.
;;    [Condition of type MY-DIVISION-BY-ZERO]


(warn 'my-division-by-zero :dividend 3) ;; no debugger


(error “This is an error!”) ;; type simple-error
```

Exception Handling

After we define our conditions, we can handle them in many ways:

- Ignore: ignore-errors
  - Returns NIL and condition
- Catch: handler-case
  - Similar to try/catch
  - General or specific
- Mapping: handler-bind
  - Specify different functions for possible conditions
- “Finally”: unwind-protect
  - Similar to the “finally” of try/catch/finally

```
(ignore-errors
  (/ 3 0))
; (condition details display here)
NIL
#<DIVISION-BY-ZERO {1008FF5F13}>
```

```
(handler-case (/ 3 0)
  (error (c)
    (format t "We caught a condition.~&")
    (values 0 c)))

(handler-case (/ 3 0)
  (division-by-zero (c)
    (format t "Caught division by zero: ~a~%" c)))
```

```
(handler-bind ((opts:unknown-option #'unknown-option)
               (opts:missing-arg #'missing-arg)
               (opts:arg-parser-failed #'arg-parser-failed))
   (opts:get-opts))
```

```
(unwind-protect (/ 3 0)
  (format t "This won’t cause issues.~&"))
```

Exception Handling

We can also use restarts and assertions to deal with conditions.

- Restarts: options in debugger used to handle conditions
  - Can define our own cases using restart-case
- Assertions: check truth value using assert and debug if needed

```
Type HELP for debugger help, or (SB-EXT:EXIT) to exit from SBCL.

restarts (invokable by number or by possibly-abbreviated name):
  0: [ABORT] Exit debugger, returning to top level.
```

```
(defun divide-with-restarts (x y)
  (restart-case (/ x y)
    (return-zero ()
      0)
    (divide-by-one ()
      (/ x 1))))
 (divide-with-restarts 3 0)
```

```
;; simplified version

restarts:
  0: [RETURN-ZERO]
  1: [DIVIDE-BY-ONE]
  2: [ABORT]
  3: [RETRY]
```

```
(assert (realp 3))
;; NIL = passed

(defun divide (x y)
  (assert (not (zerop y))
          (y)   ;; list of values we can change.
          "Y can not be zero. Please change it")
   (/ x y))
```

```
(divide 3 0)
;; Y can not be zero. Please change it
;;     [Condition of type SIMPLE-ERROR]
;;
;; Restarts:
;;  0: [CONTINUE] Retry assertion with new value for Y.
;;  …
```

Resources

1. Programming Languages Pragmatics (class textbook)
2. https://ccrma.stanford.edu/courses/220b-winter-2005/topics/commonlisp/arguments.html
3. https://lispcookbook.github.io/cl-cookbook/error_handling.html
4. http://cl-cookbook.sourceforge.net/functions.html
5. https://en.wikipedia.org/wiki/Defun
6. https://www.cs.cmu.edu/Groups/AI/html/cltl/clm/node81.html
7. http://www.gigamonkeys.com/book/functions.html
8. http://www.gigamonkeys.com/book/practical-a-simple-database.html
9. https://towardsdatascience.com/a-swift-introduction-to-common-lisp-16a2f154c423
10. https://lispcookbook.github.io/cl-cookbook/clos.html
11. http://www.gigamonkeys.com/book/object-reorientation-generic-functions.html
12. https://dept-info.labri.fr/~strandh/Teaching/MTP/Common/David-Lamkins/chapter14.html
13. https://lispcookbook.github.io/cl-cookbook/data-structures.html
14. https://www.cs.cmu.edu/Groups/AI/html/cltl/clm/node169.html