Programming Languages
A.Y. 2024-2025

Lisp and Functional Programming (5)

Laurea Triennale di Informatica
Dipartimento di Informatica, Sistemistica e Comunicazione

Marco Antoniotti marco.antoniotti@unimib.it
Fabio Sartori fabio.sartori@unimib.it

Interaction with the Common Lisp Environment

- The Lisp environment, or better its *command-line*, performs three fundamental operations; now that we know a few more things about I/O in Common Lisp we can look at them more carefully

- **Reads** (`READ`) what is typed as input
  - What is read is internally represented in appropriate data structures (numbers, characters, symbols, strings, cons-cells, and more…)
- The internal representation is **evaluated** (`EVAL`) with the goal to produce a value (or more values)
  - We will see what the `EVAL` function does
  - That is, we will write it directly in Common Lisp

- The value thus obtained is **written** (`PRINT`) out

- This is the **READ-EVAL-PRINT Loop** (**REPL**)

Expression (and Function) Evaluation: `apply` ed `eval`

- Given that programs and sexp's are equivalent in Lisp, we can state the following evaluation rules and implement them in the `eval` function.
- Given a sexp
  - If it is an atom (i.e., if it is not a cons-cell)
    - If it is a number, return its value
    - If it is a string return it as-is
    - If it is a symbol
      - Fetch the value it is associated to in the *current environment* and return it
      - If no associate value can be found, signal an error
  - If it is a cons-cell $(O\ A_1\ A_2\ \ldots\ A_n)$ then proceed as follows
    - If O is a special operation, then the list $(O\ A_1\ A_2\ \ldots\ A_n)$ is evaluated in a special way
    - If O is a symbol denoting a function in the *current environment*, then the function is applied (`apply`) to the list $(VA_1\ VA_2\ ..\ VA_n)$ that contains the values resulting from the evaluation of expressions $A_1, A_2, \ldots, A_n$
    - If O is a *Lambda Expression* it is applied to the list $(VA_1\ VA_2\ ..\ VA_n)$ that contains the values resulting from the evaluation of expressions $A_1, A_2, \ldots, A_n$
    - Otherwise, an error is signaled

The `apply` and `eval` Functions

- The `apply` function is defined as

  `apply : function list` $\rightarrow$ `sexp`

  that is, it takes a *function designator* (i.e., a symbol, a lambda-expression or a function) and it returns a value

- The `eval` function returns the value denoted by a sexp

  `eval : sexp env` $\rightarrow$ `sexp`

The `apply` and `eval` Functions

- The `apply` and `eval` functions can be directly rewritten in Common Lisp (or Scheme)

- I.e., given that in Lisp data and programs are the same thing it is possible to easily write an **interpreter** Lisp (or Scheme) in Lisp (or Scheme)
  - These types of interpreters are said **meta-circular interpreters**
  - The construction of meta-circular interpreter variants is one of the ways by which we can explore different programming modalities

The `apply` and `eval` Functions

- Let's now build the **`evaluate`** function (**`eval`** is standard) starting from the evaluation rules we just defined (*)
- The **`evaluate`** function takes a S-expression `sexp` and an environment env
  - We will soon see what an "environment" is
- The **`evaluate`** function proceeds in the following way
- **Case 1**: is `sexp` a self-evaluating function?

```
(self-evaluating-p sexp)
```

  - If yes, just return its value, that is, the Sexp
  - If not, then…

(*) The **`evaluate`** *function almost behaves as in Scheme; common Lisp has slightly different evaluation rules.*

The `apply` and `eval` Functions

- **Case 2**: is `sexp` a variable?

```
(variable-p sexp)
```

  - If yes, then find out the value bound to it
    - Where?
    - In env

```
(var-value sexp env)
```

  - If not, then …

The `apply` and `eval` Functions

- **Case 3**: is `sexp` a quoted expression like `(quote <e>)`?

  ```
  (quoted-exp-p sexp)
  ```

  - If yes, then return `<e>` as is
  - If no, then …

The `apply` and `eval` Functions

- **Case 5**: is sexp a lambda-expression? That is, a list of the form `(lambda (…) …)`?

  ```
  (lambda-exp-p sexp)
  ```

  - If yes, then create a *closure* remembering the environment in which this lambda expression is being evaluated (i.e., remembering the *static link*)

    ```
    (make-fun (lambda-exp-vars sexp)
              (lambda-exp-body sexp)
              env)
    ```

  - If no, then …

The apply and eval Functions

- **Case 9**: is sexp a function application to some arguments?

```
(application-exp-p sexp)
```

  - If yes, then **apply** the operator to the list of values obtained by evaluating every argument

```
(application (evaluate (operator sexp) env)
             (list-of-values (operands sexp) env))
```

  - If not, then …

Digression: Sequences of Evaluations in Lisp (or Scheme)

- Let's remember that **`let`** is nothing but syntactic sugar, "hiding" the application of an anonymous function
- Sequences of evaluations are also representable as subsequent function applications
- **Example**

Execution order

```
(defun foo (x)
   ((lambda (anything-here)
       (bar (1+ x)))                  ; 2
    (format t "Calling FOO (~S)~%" x))) ; 1
```

The execution of **`foo`** first evaluates the call to **`format`**, whose return value is bound to **`anything-here`**, which is **ignored** during the execution of
`(bar (1+ x))`

Digression: Sequences of Evaluations in Lisp (or Scheme)

- This idiom is so useful that it is rewritten as

  ```
  (progn <e1> <e2> … <eN>)
  ```

  or

  `(begin <e1> <e2> … <eN>)` in Scheme

- Therefore, `foo` can be rewritten as

  ```
  (defun foo (x)
     (progn
        (format t "Calling FOO (~S)~%" x)
        (bar (1+ x))))
  ```

Digression: Sequences of Evaluations in Lisp (or Scheme)

- When `progn` is the main expression in a `defun` (and in other Lisp forms too) then it can be "hidden" without problems; the `defun` body is said to be an *implicit* `progn`

- Therefore, `foo` can be finally rewritten as

```
(defun foo (x)
   ;; Implicit 'progn'.
   (format t "Calling FOO (~S)~%" x)
   (bar (1+ x))
```

The `apply` and `eval` Functions

- Let's have a look at some of the functions we assumed existing
- `self-evaluating-p`
  ```
  (defun self-evaluating-p (x) (and (atom p) (not (symbolp x))))
  ```
- `quoted-exp-p`
  ```
  (defun quoted-exp-p (x) (and (consp x) (eq (first x) 'quote)))
  ```
- `lambda-exp-p`
  ```
  (defun lambda-exp-p (x) (and (consp x) (eq (first x) 'lambda)))
  ```
- `list-of-values`
  ```
  (defun list-of-values (seq env) (mapcar (lambda (s) (evaluate s env)) seq))
  ```

The `apply` and `eval` Functions: Let's Put Everything Together

The **evaluate** function (**eval** is standard) can be built starting from the evaluation rules just described (*), plus other

```
(defun evaluate (sexp &optional (env *the-global-environment*))
   (cond ((self-evaluating-p sexp) sexp)
         ((variable-p sexp) (var-value sexp env))
         ((quoted-exp-p sexp) (exp-of-quotation sexp))
         ((if-exp-p sexp) (evaluate-if sexp env))
         ((lambda-exp-p sexp)
          (make-fun (lambda-exp-vars sexp) (lambda-exp-body sexp) env))
         ((sequence-exp-p sexp)
          (evaluate-seq (sequence-expressions sexp) env))
         ((cond-exp-p sexp) (evaluate (cond-to-if sexp) env)
         ((definition-exp-p sexp) (evaluate-def sexp env))
         ((application-exp-p sexp)
          (application (evaluate (operator sexp) env) (list-of-values (operands sexp) env)))
         (t (error ";;; Cannot evaluate : ~S." sexp))))
```

*(*) The **evaluate** function behaves like in Scheme; Common Lisp has slightly different rules*

The apply and eval Functions: Reprise

- Let’s now write the `application` function (`apply` is standard) starting for the evaluation rules previously defined (*)

```
(defun application (fun arguments)
   (cond ((primitive-fun-p fun)
          (apply-primitive-fun fun arguments))
         ((fun-p fun)
          (evaluate-seq (fun-body fun)
                        (extend-environment (fun-parameters fun)
                                            arguments
                                            (fun-environment fun))))
         (t
          (error ”Unknown function ~S in APPLICATION.” fun))))
```

- The **`application`** function calls the **`evaluate-seq`**, function, which calls **`evaluate`**; i.e., **`evaluate`** and **`application`** (**`eval`** and **`apply`**) are mutually recursive.

*(*) The **`apply`** function almost behaves as in in Scheme; Common Lisp has slightly different evaluation rules*

The Internal Representation of “Functions”

- The evaluation of a LAMBDA expression generates a function that is represented in the environment as a specialized structure (cf., the **`make-fun`** function called within **`evaluate`**) called **closure**

- This structure contains the body (the “code”) of the LAMBDA expression, the list of formal parameters, and a link to the environment where the LAMBDA expression was built
  - That is, the structure contains the ***static link*** to the evaluation environment, where it can look up the values of the free variables in the LAMBDA body

- The **`application`** function uses the *static link* contained in the closure

Environments

- The functions **`eval`** and **`apply`** rely on the implementation of **environments**, that is, on the manipulation of maps (!) associating symbols and values; an **environment** is a sequence of **frames**
- The **`var-value`** function is nothing but a 'get' of the value associated to a key in a map
- How can we implement the environment handling functions in Common Lisp?
  - **`make-frame`**
  - **`extend-env`**
  - **`var-value`**
  - **`var-value-in-frame`**

Environments

- Let’s start from the manipulation of a frame
- A frame is represented as a list of “pairs” prefixed by the **`frame`** symbol

```
(defun make-frame (vars values)
   (cons ’frame (mapcar ’cons vars values))) ; mapcar works on more than one list!
```

- **Examples**

```
cl-prompt> (make-frame ’(x y z) ’(0 1 0))
(FRAME (X . 0) (Y . 1) (Z . 0))


cl-prompt> (make-frame ’(q w) ’(the-symbol-q ”W”))
(FRAME (Q . THE-SYMBOL-Q) (W . ”W”))
```

Environments

- An environment is extended (read: an “activation frame” is pushed onto it) as follows

```
(defun extend-env (vars vals &optional (base-env *the-empty-env*))
  (if (= (length vars) (length vals))
      (cons (make-frame vars vals) base-env)
      (if (< (length vars) (length vals))
          (error "Too many arguments supplied")
          (error "Too few arguments supplied"))))
```

- We define the following as the “empty” environment

```
(defparameter *the-empty-env* ’((frame (nil . nil) (t . t))))
```

- **Examples**

```
cl-prompt> (defparameter env1 (extend-env ’(x y z) ’(0 1 0) ()))
ENV1

cl-prompt> (extend-env ’(q w) ’(the-symbol-q ”W”) env1)
((FRAME (Q . THE-SYMBOL-Q) (W . ”W”)) (FRAME (X . 0) (Y . 1) (Z . 0)))
```

Expression Rewriting

- One of the most important operations that an interpreter/compiler does is to **rewrite** an expression as a (simpler) one, in order to reuse code

- The Lisp data structures representing the expressions that `application` and `evaluate` use make this operation straightforward

  - **`COND`** is rewritten as nested **`IF`**s
  - **`LET`** / **`LET*`** / **`BEGIN`** / **`PROGN`** may be rewritten as the corresponding **`LAMBDA`** expressions

- The rewritten expression is then passed again to `evaluate` to complete the process

Meta Circular Interpreter

- The `evaluate` and `apply` functions are the pillars of the meta-circular interpreter (“interprete meta circolare” – IMC) written in Common Lisp

- IMC is distributed on the Moodle site and is split into three files
  - `env.lisp`
    contains the functions to manipulate frames and environments
  - `imc.lisp`
    contains the interpreter proper; that is the pair **`evaluate/application`** and various supporting operations, including rewriting operations
  - `repl.lisp`
    contains a simple **read-eval-print** loop (without error checking)

- All the operations previously described are contained (in one form or another) in the code

Conclusions (there is more)

- (Common) Lisp is one of the most important examples of **functional languages**
- The functional programming style, based on function composition and on the treatment of functions as first-class objects is extremely important
- Common Lisp is a much larger language than what we have seen; in particular, we have *not* seen yet
  - Imperative features
    - Assignments: `setf`
    - Iteration constructs: `dolist`, `dotimes`, `do`, `do*`, `loop`
  - Object Oriented features
    - CLOS
    - Multimethods (only Common Lisp, Dylan, Chill, R, **Julia** and few other languages have this feature "out of the box")
  - I/O
    - `open`, `close`, `write`
  - Macros!!!!
  - Exception handling
    - `error`, `handler-case`, `invoke-restart`, `catch`, `throw`
- The language is extremely flexible and can be used practically everywhere

Imperative Features in CL

Assignments, Iteration, Vectors and Arrays

Imperative Features in Common Lisp

- Lisps have had **imperative** features since its first incarnation (Don’t use them!!!!)

- The issue is how to deal with variables (and “places”) and their values, i.e., how to set them

- In Common Lisp you have the following
  - **`set`** (ancient) for names (symbols)
  - **`setq`** for names
  - **`setf`** for places

Imperative Features in Common Lisp: Assignments

- **Examples**
  - **`set`** (ancient) for names (symbols)

    ```
    cl-prompt> (set 'qd 42)
    42
    ```

  - **`setq`** for names

    ```
    cl-prompt> (setq some-list (list 1 2 3 4))
    (1 2 3 4)
    ```

  - **`setf`** for places

    ```
    cl-prompt> (setf (nth 2 some-list) 42)
    42
    ```

Imperative Features in Common Lisp: Iteration

- The first Lisps had **imperative** features loke **go-to** (named **go** in Common Lisp) to be called within **prog** blocks; this replicated (replicates) the Fortran control flow facilities, hence go-to based loops
- CL has iteration constructs
- Some have a simple and intuitive syntax and sematics
    - **`dotimes`**
      binds a variables from 0 until, but excluding, the bound

      ```
      (dotimes (i 5) (print i))
      ```

    - **`dolist`**
      binds a variable to each element of a list

      ```
      (dolist (e (list 1 2 3 4 5)) (print i))
      ```

Imperative Features in Common Lisp: Iteration

- Other iteration constructs have a more complex syntax and sematics
  - **do/do***
    bind variables to an initial value, declares a step for them, runs a test to see whether to continue and if so, executes the forms in the body; an example is below (try it)

```
(do ((j 0 (+ j 1)))
    (nil) ; Do forever.
  (format t "~%Input ~D: " j)
  (let ((item (read)))
    (if (null item)
        (return) ; Process items until NIL seen. return exists the loop.
        (format t "~&Output ~D: ~S" j item))))
```

Imperative Features in Common Lisp: Iteration

The main and more complex iteration macro is **loop**
the syntax and semantics of loop is quite complex, being a sublanguage in itself; two simple examples are the following

```
;;; Same as the previous 'do'.

(loop for j from 0
      for item = (progn (format t "~%Input ~D: " j) (read))
      while item do
        (format t "~&Output ~D: ~S" j item))))

;;; Collecting things.

(loop for e in (list 1 2 3)
      collect (+ e 42))
```

Vectors and Arrays

- Common Lisp has **vectors** and proper, multi-dimensional **arrays**
- The constructors for vectors and arrays are `vector` and `make-array`; the accessors (which refer to "places", that is "l-values" that can be assigned to) are `svref` and `aref`
- **Examples**

```
prompt> (vector 1 2 3 4)
#(1 2 3 4) ; Note the syntax with the #.

prompt> (aref (vector -2 -1 0 1 2) 2)
0

prompt> (defparameter iy (vector 0 1 0))
IV

prompt> (setf (aref iv 1) (* 42 (aref iv 1))
42

prompt> iv
#(0 42 0)
```

Vectors and Arrays

- Let's define a few of functions dealing with 2D matrices
- **Examples**

```
(defun matrix-2d (&optional (a 0) (b 0) (c 0) (d 0))
    (make-array '(2 2) :initial-contents (list (list a b) (list c d))))
```

- We can then use it as follows

```
prompt> (defparameter idmat (matrix 1 0 0 1))
IDMAT

prompt> idmat
#2A((1 0) (0 1)) ; Note what follows the '#' character: it tells you that this is is 2-dimensional array.
```

Vectors and Arrays

- Now that we have the iteration constructs we can be on more familiar ground

```
(defun mmult (m1 m2)
    ;; M1 and M2 are matrices, i.e., 2D arrays.
    ;; Of course, the following could be written with three recursive functions.
    (let ((r (matrix-2d 0 0 0 0)))
       ;; Using loop to show the general scheme.
       (loop for i from 0 below 2 do
             (loop for j from 0 below 2 do
                    (loop for k from 0 below 2 do
                          (setf (aref r i j) ; `incf' could be used instead
                                (+ (aref r i j) (* (aref m1 i k) (aref m2 k j))))
                          ) ; loop 3
                    ) ; loop 2
             ) ; loop 1
        r))
```

- We can then use it as follows

```
prompt> (mmult idm (matrix 1 2 3 4))
#2A((1 2) (3 4))
```

Vectors and Arrays

- Vectors and arrays have a large predefined interface API
- Vectors are also **sequences** (as are lists and strings)
- Arrays can be “shared”, or “displaced”

- Some consequences

```
prompt> (every 'oddp #(1 3 5 7))
T

prompt> (map 'list 'oddp #(1 3 5 7))
(T T T T)

prompt> (map 'string 'char-downcase "CamelCaseAsInJava")
"camelcaseasinjava"
```

Hash Tables

- In Common Lisp you also have hash tables that map keys to values
- The main operations are **`make-hash-table`**, **`gethash`**, **`remhash`** and **`clrhash`**
- Several *iteration* and *mapping* constructs are available to traverse a hash table
- **Examples**

```
prompt> (defparameter movies-table (make-hash-table))
MOVIES-TABLE

prompt> movies-table
#<EQL Hash Table{0} 801009CD93>

prompt> (gethash movies-table 2024)
NIL    ; The value that may be associated to 2024.
NIL    ; The confirmation that the value returned is actually correct -- it is not in this case.

prompt> (setf (gethash movies-table 2024) “Juror #2”) ; gethash is a place.
“Juror #2”
```

Hash Tables

- **More examples**

```
prompt> (gethash movies-table 2024)
"Juror #2"      ; The value found.
T               ; Yes, it is there.

prompt> (setf (gethash movies-table 1980) "The Blues Brothers")
"The Blues Brothers"

prompt> (maphash (lambda (k v) (format t "~S ==> ~S~%" k v)) movies-table)
2024 ==> "Juror #2"
1980 ==> "The Blues Brothers"
NIL

prompt> (loop for k being the hash-key of movies-table collect k)
(2024 1980)
```

Exceptional Situation Handling

- Common Lisp has – still – the most sophisticated **exceptional (error) situation handling** facilities
- There is a standard, extensible, hierarchy of **conditions**, classified as simple and serious, most of which are also **errors**
- **Example**

```
prompt> (/ 42 0)

Error: Division-by-zero caused by / of (42 0).
  1 (continue) Return a value to use.
  2 Supply new arguments to use.
  3 (abort) Return to top loop level 0.

Type :b for backtrace or :c <option number> to proceed.
Type :bug-form "<subject>" for a bug report template or :? for other options.

CL-USER 11 : 1 >
```

This is the DIVISION-BY-ZERO error

These are the "restarts"

Exceptional Situation Handling

- There are facilities to handle errors and facilities to handle restarts
- The simplest one is `handler-case`, which is akin to `try {…} catch …` in Java, C++, etc
- `handler-case` takes one form and a set of "handler clauses"
- **Example**

```
prompt> (handler-case (/ 42 0)
          (division-by-zero (dbz) -42))
-42

prompt> (handler-case (/ 42 0)
          (division-by-zero (dbz) -42)
          (t (e) 1024))
-42
```

Exceptional Situation Handling

- **Example**

```
prompt> (handler-case (/ 42 "0")
          (division-by-zero (dbz) -42)
          (t (e) 1024))
1024

prompt> (handler-case (/ 42 "0")
          (division-by-zero (dbz) -42)
          (t (e) e))
#<CONDITIONS:ARITHMETIC-TYPE-ERROR 80101B3FFB>
```

Finally: Conclusion

- Common Lisp is a large, omni-comprehensive language
- There are still some things that were left out this initial presentation
  - Object system (CLOS)
  - Generic functions and Methods
  - Packages
- It is (mostly) very, very well designed with interleaving and interconnecting parts
- There are is several interesting Common Lisp related projects that can be pursued as a LT stage (or LM thesis)

’(Happy Hacking)