Meta-programming in Common Lisp

Damien Cassou

Software Architecture Group

Syntax

```
(function-name arg1 arg2 ... argn)
```

Syntax

```
(function-name arg1 arg2 ... argn)
```

```
> (+ 1 2)
3
```

Creating Lists

```
> (cons 3 nil)
(3)
```

Creating Lists

```
> (cons 3 nil)
(3)
> (cons 2 (3))
```

Creating Lists

```
> (cons 3 nil)
(3)
> (cons 2 (3))
```

Impossible as 3 is not a function

Creating Lists

```
> (cons 3 nil)
(3)
> (cons 2 (3))
```

Impossible as 3 is not a function

```
> (cons 2 '(3))
(2 3)
```

Creating Lists

```
> (cons 3 nil)
(3)
> (cons 2 (3))
```

Impossible as 3 is not a function

```
> (cons 2 '(3))
(2 3)
> (cons 1 '(2 3))
(1 2 3)
```

Creating Lists

```
> (cons 3 nil)
(3)
> (cons 2 (3))
```

Impossible as 3 is not a function

```
> (cons 2 '(3))
(2 3)
> (cons 1 '(2 3))
(1 2 3)
> (list 1 2 3)
(1 2 3)
```

Creating Lists

```
> (cons 3 nil)
(3)
> (cons 2 (3))
```

Impossible as 3 is not a function

```
> (cons 2 '(3))
(2 3)
> (cons 1 '(2 3))
(1 2 3)
> (list 1 2 3)
(1 2 3)
> '(1 2 3)
(1 2 3)
```

Studying Lists

```
> (car '(1 2 3))
1
```

Studying Lists

```
> (car '(1 2 3))
1
> (cdr '(1 2 3))
(2 3)
```

Studying Lists

```
> (car '(1 2 3))
1
> (cdr '(1 2 3))
(2 3)
> (first '(1 2 3))
1
```

Studying Lists

```
> (car '(1 2 3))
1
> (cdr '(1 2 3))
(2 3)
> (first '(1 2 3))
1
> (last '(1 2 3) 2)
(2 3)
```

Studying Lists

```
> (car '(1 2 3))
1
> (cdr '(1 2 3))
(2 3)
> (first '(1 2 3))
1
> (last '(1 2 3) 2)
(2 3)
> (last '(1 2 3))
(3)
```

Creating Functions

```
> (defun mult2 (x)
    "Multiplies x by 2"
    (* x 2))
mult2
```

Creating Functions

```
> (defun mult2 (x)
    "Multiplies x by 2"
    (* x 2))
mult2
```

defun is itself a function, it creates functions

Creating Functions

```
> (defun mult2 (x)
    "Multiplies x by 2"
    (* x 2))
mult2
```

defun is itself a function, it creates functions

```
> (mult2 3)
6
```

Studying Functions

```
> (describe mult2)
```

Studying Functions

```
> (describe mult2)
```

Impossible because mult2 is not a variable

Studying Functions

```
> (describe mult2)
```

Impossible because mult2 is not a variable

```
> #'mult2
#<FUNCTION mult2>
```

Studying Functions

```
> (describe mult2)
```

Impossible because mult2 is not a variable

```
> #'mult2
#<FUNCTION mult2>
> (describe #'mult2)
(defun mult2 (x)
  "Multiplies x by 2"
  (* x 2))
```

Calling Functions

```
>  (mult2  3)
6
```

Calling Functions

```
> (mult2 3)
6
> (funcall #'mult2 3)
6
```

Calling Functions

```
> (mult2 3)
6
> (funcall #'mult2 3)
6
> (defvar fmult2 #'mult2)
fmult2
```

Calling Functions

```
> (mult2 3)
6
> (funcall #'mult2 3)
6
> (defvar fmult2 #'mult2)
fmult2
> (funcall fmult2 3)
6
```

Summary

In Lisp it is possible to:

- define new functions,
- retrieve a function by name,
- reference a function from a variable,
- call a function from a variable.

Summary

In Lisp it is possible to:

- define new functions,
- retrieve a function by name,
- reference a function from a variable,
- call a function from a variable.

This is very similar to pointer manipulation in C

Function Pointer Manipulation in C

```
int mult2 (int c) {
 return c * 2;
}
```

Function Pointer Manipulation in C

```
int mult2 (int c) {
 return c * 2;
}
```

```
int main(void) {
   int (*fmult2) (int) = mult2;
   (*fmult2)(3);
}
```

Generating new Functions

```
> (get-source 'mult2)
(nil nil
     (defun mult2 (x)
       "Multiplies x by 2"
       (* x 2)))
```

Generating new Functions

```
> (get-source 'mult2)
(nil nil
     (defun mult2 (x)
       "Multiplies x by 2"
       (* x 2)))
```

don't try this at home!

Generating new Functions

```
> (defvar smult2
    (third (get-source 'mult2)))
smult2
```

Generating new Functions

```
> (defvar smult2
    (third (get-source 'mult2)))
smult2
> smult2
(defun mult2 (x)
  "Multiplies x by 2"
  (* x 2))
```

Generating new Functions

```
> (first smult2)
defun
```

Generating new Functions

```
> (first smult2)
defun
> (second smult2)
mult2
```

Generating new Functions

```
> (first smult2)
defun
> (second smult2)
mult2
> (third smult2)
(x)
```

Generating new Functions

```
> (first smult2)
defun
> (second smult2)
mult2
> (third smult2)
(x)
> (fourth smult2)
"Multiplies x by 2"
```

Generating new Functions

```
> (first smult2)
defun
> (second smult2)
mult2
> (third smult2)
(x)
> (fourth smult2)
"Multiplies x by 2"
> (fifth smult2)
(* x 2)
```

Generating new Functions

```
> (defvar smult10
      (copy-list smult2))
smult10
```

Generating new Functions

```
> (defvar smult10
      (copy-list smult2))
smult10
> (nsubstitute 10 2 (fifth smult10))
nil
```

Generating new Functions

```
> (defvar smult10
      (copy-list smult2))
smult10
> (nsubstitute 10 2 (fifth smult10))
nil
> smult10
(defun mult2 (x)
  "Multiplies x by 2"
  (* x 10))
```

Generating new Functions

```
> smult10
(defun mult2 (x)
   "Multiplies x by 2"
  (* x 10))
```

Generating new Functions

```
> smult10
(defun mult2 (x)
   "Multiplies x by 2"
   (* x 10))
> (nsubstitute 'mult10 'mult2
                smult10)
(defun mult10 (x)
   "Multiplies x by 2"
   (* x 10))
```

Generating new Functions

```
> smult10
(defun mult10 (x)
  "Multiplies x by 2"
  (* x 10))
```

Generating new Functions

```
> smult10
(defun mult10 (x)
  "Multiplies x by 2"
  (* x 10))
> (setf (fourth smult10)
        (cl-ppcre:regex-replace "2"
          (fourth smult10) "10"))
"Multiplies x by 10"
```

Generating new Functions

```
> smult10
(defun mult10 (x)
  "Multiplies x by 10"
  (* x 10))
```

Generating new Functions

```
> smult10
(defun mult10 (x)
  "Multiplies x by 10"
  (* x 10))
> (eval smult10)
mult10
```

Generating new Functions

```
> smult10
(defun mult10 (x)
  "Multiplies x by 10"
  (* x 10))
> (eval smult10)
mult10
> (mult10 3)
30
```

Summary

- A function definition in Lisp is a list.
- This list can be studied like any list.
- New functions can be created from a list.

Beyond Functions

How would you implement while that executes its body *as long as* its condition stays true?

```
> (while condition body)
```

The While Construct

```
> (setq i 10)
> (while (/= i 0)
    (decf i)
    (format t "i is now: ~s~%" i))
```

The While Construct

```
> (setq i 10)
> (while (/= i 0)
     (decf i)
     (format t "i is now: ~s~%" i))
i is now: 9
i is now: 8
i is now: 7
...
i is now: 2
i is now: 1
i is now: 0
```

The While Construct: Using Loop

```
> (while (/= i 0)
    (decf i)
    (format t "i is now: ~s~%" i))
> (loop
    (if (not (/= i 0))
      (return)
      (progn
        (decf i)
        (format t "i = ~s~%" i))))
```

The While Construct: Function

```
> (while (/= i 0)
    (decf i)
    (format t "i is now: ~s~%" i))
> (defun while (test &rest body)
    (loop
      (if (not test)
          (return)
          (progn body))))
```

The While Construct: Function

```
> (while (/= i 0)
    (decf i)
    (format t "i is now: ~s~%" i))
> (defun while (test &rest body)
    (loop
      (if (not test)
          (return)
          (progn body))))
```

doesn't work because parameters are evaluated immediately

```
> (while nil nil)
```

The While Construct: Function

```
> (while '(/= i 0)
    '(decf i)
    '(format t "i is now: ~s~%" i))
```

The While Construct: Function

```
> (while '(/= i 0)
    '(decf i)
    '(format t "i is now: ~s~%" i))
> (defun while (test &rest body)
    (loop
      (if (not (eval test))
          (return)
          (mapcar #'eval body))))
```

The While Construct: Function

```
> (while '(/= i 0)
    '(decf i)
    '(format t "i is now: ~s~%" i))
> (defun while (test &rest body)
    (loop
      (if (not (eval test))
          (return)
          (mapcar #'eval body))))
```

works, but using while is less readable than intended

Summary

- Arguments of functions are evaluated first.
- To prevent evaluation, use quote (or ').
- Use **eval** to evaluate an expression.

Macros

Macros are programs that write programs

- they return lists representing Lisp code.
- they don't evaluate their arguments.
- they are evaluated at *compile time*.

The While Construct: Macro

```
> (loop
    (if (not (/= i 0))
      (return)
      (progn
        (decf i)
        (format t "i = ~s~%" i))))
```

The While Construct: Macro

```
> (loop
    (if (not (/= i 0))
      (return)
      (progn
        (decf i)
        (format t "i = ~s~%" i))))
> (defmacro while (test &body body)
    (list 'loop
      (list 'if (list 'not test)
        (list 'return)
        (cons 'progn body))))
```

The While Construct: Macro

```
> (loop
    (if (not (/= i 0))
      (return)
      (progn
        (decf i)
        (format t "i = ~s~%" i))))
> (defmacro while (test &body body)
    `(loop
       (if (not ,test)
         (return)
         (progn ,@body))))
```

Creating an OO language

```
> (makeClass Speaker (name)
    (makeMethod speak (sentence)
      (format t
        "Listen all of you: ~s~%"
        sentence)))
```

Creating an OO language

```
> (makeClass Speaker (name)
    (makeMethod speak (sentence)
      (format t
        "Listen all of you: ~s~%"
        sentence)))
> (defvar alex
          (new 'Speaker "Alex"))
```

Creating an OO language

```
> (makeClass Speaker (name)
    (makeMethod speak (sentence)
      (format t
        "Listen all of you: ~s~%"
        sentence)))
> (defvar alex
          (new 'Speaker "Alex"))
> (call alex 'speak "Hello World!")
Listen all of you: "Hello World!"
```

Creating an OO language

```
> (makeClass Speaker (name)
    (makeMethod speak (sentence)
      (format t
        "Listen all of you: ~s~%"
        sentence)))
> (defvar alex
          (new 'Speaker "Alex"))
> (call alex 'speak "Hello World!")
Listen all of you: "Hello World!"
> (getinstvar alex 'name)
Alex
```

Creating an OO language

```
> (makeClass Speaker ()
    (makeMethod "..."))
```

Creating an OO language

```
> (makeClass Speaker ()
    (makeMethod "..."))
```

A class is composed of:

- a name,
- some instance variables,
- and some method definitions.

Creating an OO language

```
> (makeClass Speaker ()
    (makeMethod "..."))
```

A class is composed of:

- a name,
- some instance variables,
- and some method definitions.

```
> (defstruct cls
    name
    vars
    mths)
```

Creating an OO language

```
> (makeClass Speaker ()
    (makeMethod "..."))
```

```
> (defmacro makeClass (name iVars
                       &body meths)
    (push
       (make-cls
          :name ',name
          :vars ',iVars
          :mths
            ',(mapcar #'eval meths))
       *classes*))
```

Creating an OO language

```
>  (makeMethod speak (sentence)
     (format t "..." sentence))
```

Creating an OO language

```
> (makeMethod speak (sentence)
    (format t "..." sentence))
```

A method is composed of:

- a name,
- some parameters,
- a body

Creating an OO language

```
> (makeMethod speak (sentence)
    (format t "..." sentence))
```

A method is composed of:

- a name,
- some parameters,
- a body

```
> (defstruct mth
    name
    lmbd)
```

Creating an OO language

```
> (makeMethod speak (sentence)
    (format t "..." sentence))
```

Creating an OO language

```
> (makeMethod speak (sentence)
    (format t "..." sentence))
```

```
> (defmacro makeMethod (name
                        argNames &body body)
    `(make-mth
        :name ',name
        :lmbd (lambda ,argNames
                      ,@body)))
```

Creating an OO language

```
> (new 'Speaker "Alex")
```

Creating an OO language

```
> (new 'Speaker "Alex")
```

An object is composed of:

- a reference to its class,
- some values for its instance variables

Creating an OO language

```
> (new 'Speaker "Alex")
```

An object is composed of:

- a reference to its class,
- some values for its instance variables

```
> (defstruct obj
    cls
    values)
```

Creating an OO language

```
> (call alex 'speak "Hello World!")
Listen all of you: "Hello World!"
```

Creating an OO language

```
> (call alex 'speak "Hello World!")
Listen all of you: "Hello World!"
```

A call is a function with:

- the receiver object,
- a method to be executed,
- and a list of parameters.

Creating an OO language

```
> (call alex 'speak "Hello World!")
Listen all of you: "Hello World!"
```

A call is a function with:

- the receiver object,
- a method to be executed,
- and a list of parameters.

```
(defun call (obj name &rest params)
   "...")
```

Creating an OO language

```
(defun call (obj name &rest params)
  (let* ((cls (obj-cls obj))
         (mth (getMethod cls name)))
    (apply (mth-lmbd mth)
           params)))
```

Creating an OO language

```
(defun call (obj name &rest params)
  (let* ((cls (obj-cls obj))
         (mth (getMethod cls name)))
    (apply (mth-lmbd mth)
           params)))
```

```
(defun getMethod (cls name)
  (find name (cls-mths cls)
        :key #'mth-name))
```

Creating an OO language

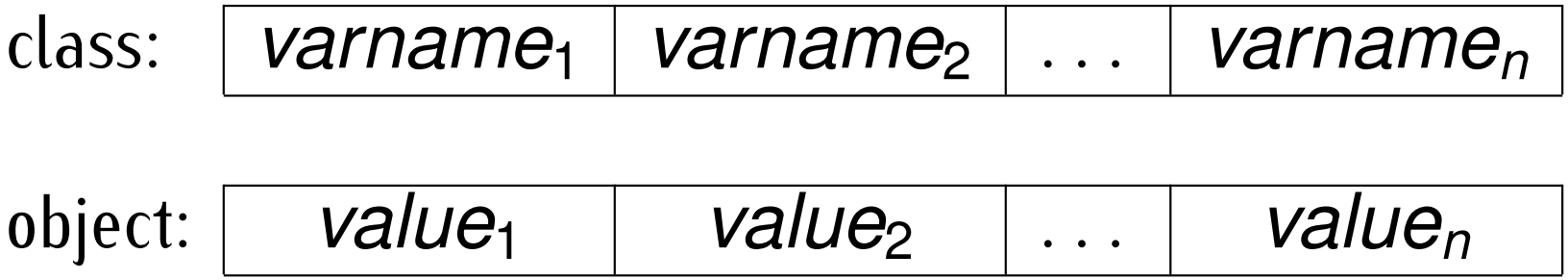
```
> (getinstvar alex 'name)
Alex
```

Creating an OO language

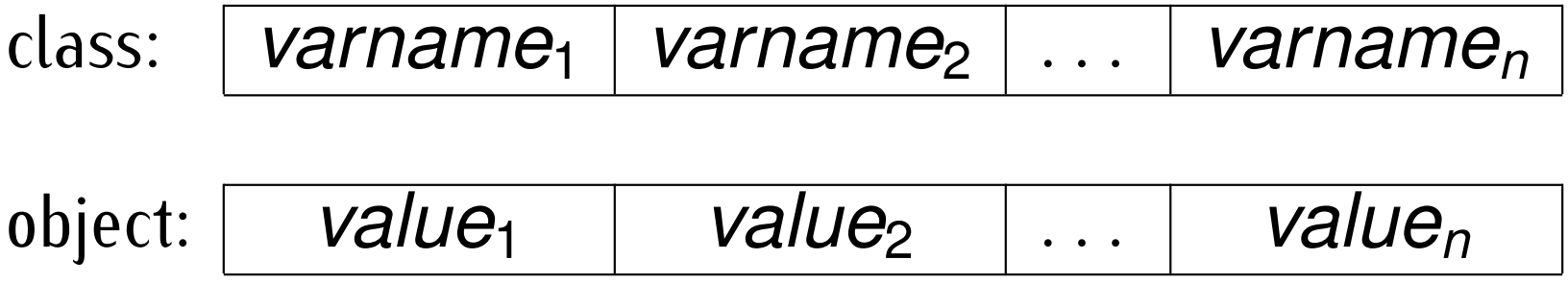
```
> (getinstvar alex 'name)
Alex
```

Looking for an instance variable value from its name involves:

- getting the position of the name in the list of all instance variables of the class,
- taking the value at this position in the list of all values of the object.



Creating an OO language




```
(defun getInstVar (obj name)
  (let* ((cls (obj-cls obj))
         (vars (cls-vars cls))
         (pos (position name vars)))
    (nth pos (obj-values obj))))
```

Handling this

An object must be able to get its instance variables and call methods by using this .

Handling this

An object must be able to get its instance variables and call methods by using this .

```
> (makeClass Speaker (name)
    (makeMethod getName ()
      (getInstVar 'this 'name)))
```

Handling this

An object must be able to get its instance variables and call methods by using this .

```
> (makeClass Speaker (name)
    (makeMethod getName ()
      (getInstVar 'this 'name)))
> (call alex 'getname)
Alex
```

Handling this

An object must be able to get its instance variables and call methods by using this .

```
> (makeClass Speaker (name)
    (makeMethod getName ()
      (getInstVar 'this 'name)))
> (call alex 'getname)
Alex
```

This requires the system to keep track of the *current object*.

Handling this

An object must be able to get its instance variables and call methods by using this .

```
> (makeClass Speaker (name)
    (makeMethod getName ()
      (getInstVar 'this 'name)))
> (call alex 'getname)
Alex
```

This requires the system to keep track of the *current object*.

```
> (defparameter *cur-obj* nil)
```

Handling this

```
(defun getInstVar (obj name)
  (let* ((theObj
            (if (equal obj 'this)
                *cur-obj*
                obj))
         (cls (obj-cls theObj))
         (vars (cls-vars cls))
         (pos (position name vars)))
    (nth pos (obj-values theObj))))
```

Handling this

```
(defun getInstVar (obj name)
  (let* ((theObj
            (if (equal obj 'this)
                *cur-obj*
                obj))
         (cls (obj-cls theObj))
         (vars (cls-vars cls))
         (pos (position name vars)))
    (nth pos (obj-values theObj))))
```

When is *cur-obj* updated?

Handling this

```
(defun getInstVar (obj name)
  (let* ((theObj
            (if (equal obj 'this)
                *cur-obj*
                obj))
         (cls (obj-cls theObj))
         (vars (cls-vars cls))
         (pos (position name vars)))
    (nth pos (obj-values theObj))))
```

When is *cur-obj* updated? Before it is *used*!

Handling this

```
(defun getInstVar (obj name)
  (let* ((theObj
            (if (equal obj 'this)
                *cur-obj*
                obj))
         (cls (obj-cls theObj))
         (vars (cls-vars cls))
         (pos (position name vars)))
    (nth pos (obj-values theObj))))
```

When is *cur-obj* updated? Before it is *used*!
As this is only used when a method is executed, the method call needs to do the updating job.

Handling this

The method call needs to do the updating job:

```
(defun call (obj name &rest params)
  (let* ((cls (obj-cls obj))
         (mth (getMethod cls name))
         (*cur-obj* obj))
    (apply (mth-lmbd mth)
           params)))
```

Handling this

As we also want to pass this as first argument to call :

```
(defun call (obj name &rest params)
  (let* ((theObj
          (if (equal obj 'this)
              *cur-obj*
              obj))
         (cls (obj-cls theObj))
         (mth (getMethod cls name)))
    (setf *cur-obj* theObj)
    (apply (mth-lmbd mth)
           params)))
```

Creating an OO language

Possible improvements:

- setting of instance variables
- inheritance
- constructors
- dedicated syntax

Creating Domain-Specific Languages

```
(makeClass Speaker (name)
  (makeMethod speak (s)
    (format t "I say: ~a" s))
  (makeMethod getName ()
    (call 'this 'speak "hi!")
    (getInstVar 'this 'name)))
```

Creating Domain-Specific Languages

```
(makeClass Speaker (name)
  (makeMethod speak (s)
    (format t "I say: ~a" s))
  (makeMethod getName ()
    (call 'this 'speak "hi!")
    (getInstVar 'this 'name)))
```

```
(makeMethod getName ()
    {c speak "hi!"}
    {i name})
```

Creating Domain-Specific Languages

```
;; {c speak "hi!"} {i name}
(set-macro-character #\{
 (lambda (str c)
   (declare (ignore c))
   (let ((type (read-char str))
         (l (read-delimited-list
                          #\} str)))
     (case type
       (#\i `(getInstVar 'this
                         ',(car l)))
       (#\c `(call 'this
                   ',(car l)
                   ,@(cdr l)))))))
```

Acknowledgments

Thanks to #lisp for all their help:

- akovalenko
- antifuchs
- H4ns
- nikodemus
- pjb
- prxb
- ThomasH