

Introduction to Cognitive Robotics

Module 8: An Introduction to Functional Programming with Lisp

Lecture 3: Lambda and mapping functions, CLOS, inference

David Vernon
Carnegie Mellon University Africa

www.vernon.eu

Lambda

- The `defun` macro creates a function and gives it a name
- However, functions don't have to have names, and we don't need `defun` to define them
- We can refer to functions literally using a `lambda expression`
- A list containing the symbol `lambda`, followed by a `list of parameters`, followed by a `body` of zero or more `expressions`, e.g.

```
(lambda (x y)
  (+ x y))
```

Lambda

- A lambda expression can be considered as the name of a function and can be the first element of a function call

```
CL-USER> ((lambda (x) (+ x 100)) 1)
```

```
101
```

```
CL-USER> (funcall #'(lambda (x) (+ x 100)) 1)
```

```
101
```

- This notation allows us to use functions without naming them
- We'll see the need for this later

Mapping Functions

- Common Lisp provides several functions for calling functions on the elements of a list, e.g. `mapcar`, `maplist`, `mapc`, and `mapcan`
- The most frequently used is `mapcar`
 - Takes a function and one or more lists
 - Returns the result of applying the function to `elements` taken from each list (until some list runs out)

```
CL-USER> (mapcar #'(lambda (x) (+ x 10)) '(1 2 3))  
(11 12 13)
```


```
CL-USER> (mapcar #'list  
                '(a b c)  
                '(1 2 3 4))  
((A 1) (B 2) (C 3))
```

Macros

- Macros are operators that are implemented by translation (i.e. text-substitution) called a **macro expansion**
- Done automatically at compile time, not at runtime
- Macros are defined using **defmacro**
- We'll mainly be using macros rather than writing them

Macros

```
CL-USER> (defmacro nil! (x) ;macro to set the argument
           (list `setf x nil)) ;to nil
CL-USER> (nil! x)
NIL
CL-USER> x
NIL
CL-USER> (macroexpand-1 '(nil! x))
(SETF X NIL)
T
```

 This is translated into (setf x nil)


CLOS – Common Lisp Object System

- Class has a name and a number of slots
- Slot data associated with a class
- Superclass a class from which other classes inherit slots
- Instance an instantiation of a class using `make-instance`
- Method a function associated with a class

CLOS – Common Lisp Object System


Examples: class definition

```
(defclass rectangle ()  
  (height width))
```




Class with two slots

```
(defclass circle ()  
  (radius center))
```




Class with two slots

```
(defclass colored ()  
  (color))
```



Class with one slot

```
(defclass colored-circle (circle colored)  
  ())
```



Class colored-circle inherits slots from two superclasses [slots radius, center and color]

CLOS – Common Lisp Object System

Examples: class methods

```
(defmethod area ((x rectangle))  
  (* (slot-value x 'height) (slot-value x 'width)))
```

```
(defmethod area ((x circle))  
  (* pi (expt (slot-value x 'radius) 2)))
```

```
CL-USER> (let ((r (make-instance 'rectangle)))  
           (setf (slot-value r 'height) 2  
                 (slot-value r 'width) 3)  
           (area r))
```

r is an instance of the class rectangle

6

slot-value provides access to the slots of a class, in this case the width slot of instance r

CLOS – Common Lisp Object System

Example: use of **keyword arguments** when defining a class

```
((defclass circle ())  
  ((radius :accessor circle-radius)  
   (center :accessor circle-center)))
```

```
CL-USER> (setf c (make-instance 'circle))
```

```
#<Circle #XC5C726>
```

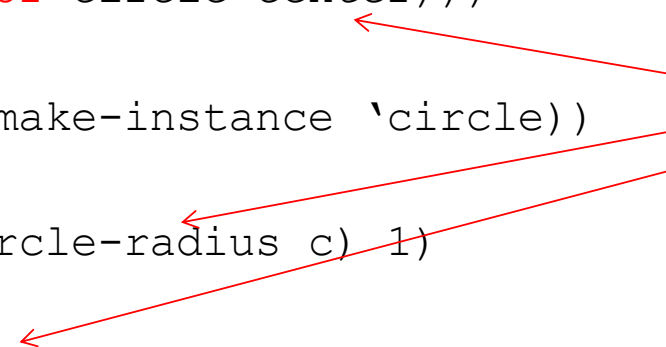
```
CL-USER> (setf (circle-radius c) 1)
```

```
1
```

```
CL-USER> (circle-radius c)
```

```
1
```

Allows access to
the slot value
without having to
use slot-value



CLOS – Common Lisp Object System

Example: use of **keyword arguments** when defining a class

```
((defclass circle ()  
  ((radius :reader circle-radius)  
   (center :writer circle-center)))
```

Allows **read access** to the slot value
without having to use `slot-value`

Allows **write access** to the slot value
without having to use `slot-value`

CLOS – Common Lisp Object System

Example: use of **keyword arguments** when defining a class

```
(defclass circle ()  
  ((radius :accessor circle-radius  
           :initarg :radius  
           :initform 1)  
   (center :accessor circle-center  
           :initarg :center  
           :initform (cons 0 0))))
```

Allows the specification of an initial value when creating an instance with **make-instance** using the `:radius` keyword

Specifies a default value for the slot

Note that `:initargs` take precedence over `:initforms`.

The default value for the center slot is a cons pair

CLOS – Common Lisp Object System


Example: use of **keyword arguments** when defining a class

```
CL-USER> (setf c (make-instance 'circle :radius 3))
#<Circle #XC2DE0E>
CL-USER> (circle-radius c)
3
CL-USER> (circle-center c)
(0 . 0)
```

CLOS – Common Lisp Object System

We can specify that a slot shares the **same value in every instance of a class** by declaring it to have `:allocation :class`

```
((defclass circle ())  
  ((radius      :accessor circle-radius)  
   (center      :accessor circle-center)  
   (show-center :allocation :class)))
```



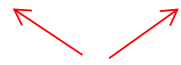
All instances of circle will either show the center point or not because they all share the same value of `show-center`

CLOS – Common Lisp Object System

Multiple inheritance (NB: this is a very shallow treatment)

- We have seen that a class can inherit from several superclasses

```
(defclass colored-circle (circle colored)
  ( ))
```



Class `colored-circle` inherits slots from two
superclasses `radius` and `color`

- A class inherits the **union** of the slots of its superclasses
 - A superclass may itself have superclasses

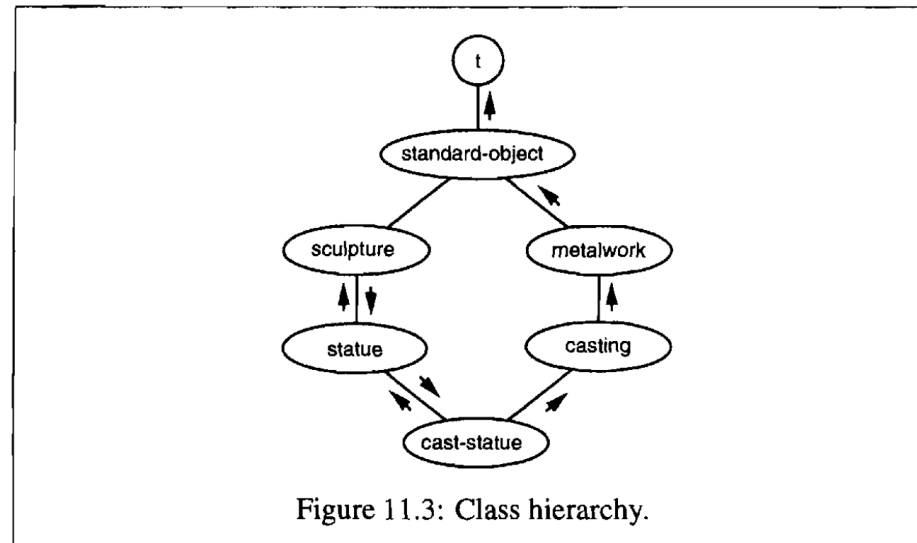
CLOS – Common Lisp Object System

Multiple inheritance (very shallow treatment)

- We represent the **class hierarchy** of inherited superclasses with a graph
- The **precedence list** of superclasses is an ordering of the class and its superclasses from **most specific** to **least specific**
 - Superclasses nearer to the class being defined are more specific
 - Superclasses farther from the class being defined are less specific
- The precedence list is determined by a **traversal** of the class hierarchy

CLOS – Common Lisp Object System

Multiple inheritance (very shallow treatment)



Credit: P. Graham, ANSI Common Lisp, Prentice-Hall, 1996

CLOS – Common Lisp Object System

Multiple inheritance (very shallow treatment)

1. When there are methods with the same name in the class hierarchy, how do we decide which one to call?

Use the **most specific method** for which the classes of the arguments match the specializations of the parameters

2. When there are slots with the same name in the class hierarchy, how do we combine the properties of the slots?

Use a single slot that **combines the properties of the slots** in the superclasses according to rules based on the precedence list

Inference

- **Facts** can be represented by a list comprising

- Predicate
- Zero or more arguments

(parent donald nancy)

donald is the parent of nancy



- **Rules** tell what can be inferred from the facts we already have

then-part

if-part

(\leftarrow head body)

"We can infer that x is the child of y if y is the parent of x "

[Alternatively, we can prove any fact of the form (child x y) by proving (parent y x)]

(\leftarrow (child ? x ? y) (parent ? y ? x))



Variables are represented as symbols beginning with a question mark

Inference

- The body (if-part) of a rule can be a complex expression
- For example, a rule that if x is the parent of y , and x is male, then x is the father of y , would be written

```
(← (father ?x ?y) (and (parent ?x ?y) (male ?x)))
```

Inference

- Rules may depend on facts implied by other rules
- The proof of an expression can continue back through any number of rules (so long as it eventually ends up using known facts)
- This is known as **backward chaining** from what we want to prove to what we already know
- Inference in CRAM is handled by a built-in **Prolog interpreter** (written in Lisp) in the `cram_reasoning` package

Packages

- Large programs are often divided up into multiple packages
- Packages provide the equivalent of a namespace in other languages
 - A symbol defined in one package is local to that package
 - A symbol has to be explicitly **exported** to be visible in another package
 - An exported symbol usually has to be **qualified** by in the package using it by preceding it with the name of the package that owns it

Packages

- For example, suppose a program is divided into two packages `math` and `disp`
- If the symbol `fft` is exported by the `math` package
 - Code in the `disp` package will be able to refer to it as `math:fft`
 - In the `math` package, it will be possible to refer to it as simply `fft`

Packages

The following is an example of what you put at the top of a file containing a distinct package of code:

```
(defpackage "MY-APPLICATION"
  (:use "COMMON-LISP" "MY-UTILITIES")
  (:nicknames "APP")
  (:export "WIN" "LOSE" "DRAW"))

(in-package my-application)
```

Symbols exported from these packages are accessible without package qualifiers

Code in other packages can refer to these symbols as `app:win`, `app:lose`, and `app:draw`

Export these symbols: `win`, `lose`, and `draw`

Makes the current package `my-application`

Recommended Reading

P. Graham. *ANSI Common Lisp*, Prentice-Hall, 1996, Chapter 2.

<http://ep.yimg.com/ty/cdn/paulgraham/acl2.txt>

The Lisp pages on Paul Graham's website:

<http://paulgraham.com/lisp.html>

especially the following:

What Made Lisp Different:

<http://paulgraham.com/diff.html>

Revenge of the Nerds

[Essentially, the story of Lisp]

<http://paulgraham.com/icad.html>