Introduction to Cognitive Robotics

Module 8: An Introduction to Functional Programming with Lisp

Lecture 2: Common Lisp - functions, I/O, recursion, iteration

David Vernon
Carnegie Mellon University Africa

www.vernon.eu

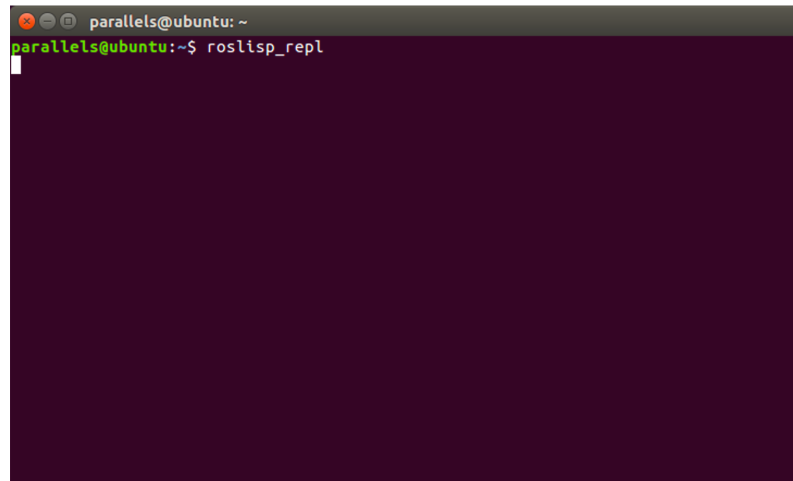
REPL
(Read-Eval-Print Loop)

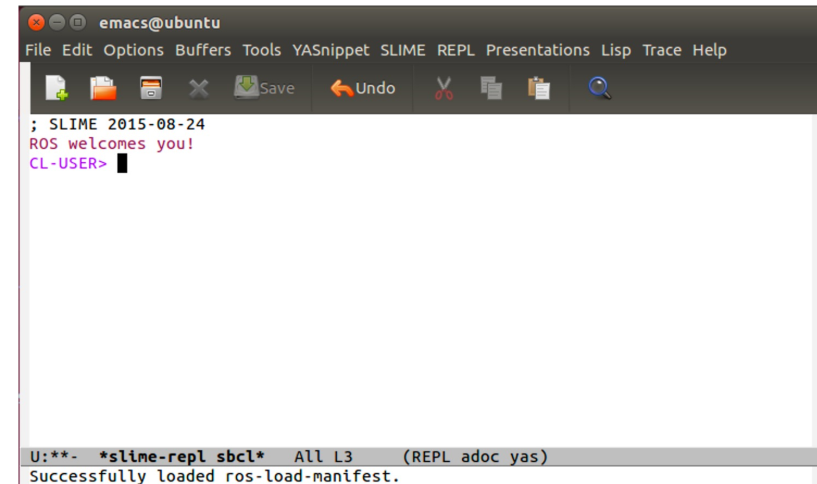
As you go through this set of slides, you might like to try out the examples.

Use `roslisp_repl` to launch the Lisp compiler's interactive front-end: REPL

```
$ roslisp_repl
```

Command entered in terminal





Functions

- You can define new functions with `defun`

- Functions usually takes three or more arguments:

  1. a name
  2. a list of parameters
  3. one or more expressions that will make up the body of the function

Parameter (used in the definition)

```
CL-USER> (defun our-third (x)
           (car (cdr (cdr x))))
OUR-THIRD
CL-USER> (our-third '(a b c d))
C
```

Argument (used in when calling)

Parameter lists

Optional parameters

- Can be omitted and a default value used instead
- Place `&optional` after the the last required parameter in the parameter list

```
CL-USER> (defun philosoph (thing &optional property)
            (list thing 'is property))
CL-USER> (philosoph 'death)
(DEATH IS NIL)
```

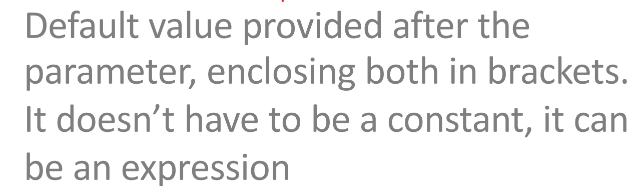
Parameter lists

Optional parameters

– Can be omitted and a default value used instead

– Place `&optional` after the the last required parameter in the parameter list

```
CL-USER> (defun philosoph (thing &optional (property 'fun))
            (list thing 'is property))
CL-USER> (philosoph 'death)
(DEATH IS FUN)
```



Parameter lists

Keyword parameters
(NB … these are used a lot in the CRAM plan language)

- Also an optional parameter but more flexible
- Place `&key` after the the last required parameter in the parameter list
- All parameters after it are optional
- When the function is called, these parameters will be identified not by their position (as is usual) but by symbolic tags that precede them

Parameter lists

Keyword parameters

(NB … these are used a lot in CRAM)

```
CL-USER> (defun keylist (a &key x y z)
           (list a x y z))
KEYLIST
CL-USER> (keylist 1 :y 2)
(1 NIL 2 NIL)
CL-USER> (keylist 1 :y 3 :x 2)
(1 2 3 NIL)
```

The parameter names are used to define the associated keywords by prepending a colon, e.g. `:x`, `:y`, and `:z`

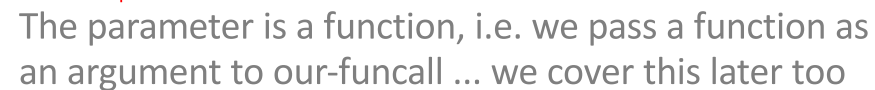
Note the arbitrary ordering of the keyword arguments

Parameter lists

Variable number of arguments

- Place `&rest` before the last variable in the parameter list
- When the function is called, this variable will be set to be a list of all the remaining arguments

```
CL-USER>(defun our-funcall (fn &rest args)
           (apply fn args))
```



Return Values

- All the functions we have seen so far return just one value

- Functions can return multiple values using the `values` function

  - It returns exactly the values you give it as arguments

```
CL-USER>(values 'a nil (+ 2 4))
A
NIL
6
```

  - For a function to return multiple values, make the `values` expression the last thing to be evaluated in the body of the function

Return Values

Functions can return multiple values using the `values` function

- The returned multiple values are accessed with the `multiple-value–bind` function

```
CL-USER> (multiple-value-bind (x y z) (values 1 2 3)
              (list x y z))
(1 2 3)
```

Instead of this expression, you would have the call to the function that returns multiple values

Return Values

Functions can return multiple values using the `values` function

- If there are more variables than values, the leftover ones will be `nil`.
- If there are more values than variables, the extra values will be discarded.

```
CL-USER> (multiple-value-bind (x y z) (values 1 2)
           (list x y z))
(1 2 NIL)
CL-USER> (multiple-value-bind (s m h) (get-decoded-time)
              (format nil "~A:~A:~A" h m s))
"4:32:13"
```

Pick up the first three values

The `format` function is covered later

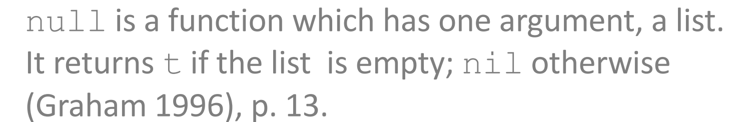
If this had been `t` the time would have been printed to the terminal

This function returns the time in nine values: second, minute, hour, date, month, day, and two others

Recursion

Function to test whether something is an element of a list

```
CL-USER> (defun our-member (obj lst)
             (if (null lst)
                    nil
                    (if (eql (car lst) obj)
                           lst
                           (our-member obj (cdr lst)))))
CL-USER> (our-member 'b '(a b c))
(B C) ;not NIL so therefore true
CL-USER> (our-member 'z '(a b c))
NIL  ;false
```



Input and Output

The `format` function

Send the output to the terminal

```
CL-USER> (format t "~A plus ~A equals ~A.~%" 2 3 (+ 2 3))
2 plus 3 equals 5.
NIL
```

a string goes here

newline

Three arguments for the ~As

Input and Output

The `read` function

```
CL-USER> (defun askem (string)
            (format t ”~A" string) ;prompt the user
            (read))
CL-USER> (askem "How old are you? ")
How old are you? 29
29
```

← The function returns the value of the last expression evaluated

Variables

- The `let` operator introduces new local variables

```
CL-USER> (let ((x 1) (y 2))
           (+ x y))
3
```

- It has two parts
  - A list of instructions for creating variables
    - Each of the form (*variable expression*)
    - Each variable will be initially set to the value of the expression
    - The variables are valid in the body of the `let` (i.e. they are local variables)
  - The body of expressions
    - Each is evaluated in order
    - The value of the last expression is returned as the value of the `let`

Variables

- The `let` operator introduces new local variables

```
(defun ask-number ()
  (format t "Please enter a number. ")
  (let ((val (read)))
    (if (numberp val)
        val
        (ask-number))))
```

```
CL-USER> (ask-number)
Please enter a number. a
Please enter a number. (ho hum)
Please enter a number. 52
52
```

Variables

- The `let*` operator introduces new local variables, the values of which can depend on each other

```
CL-USER> (let* ((x 1)
                (y (+ x 1)))
           (+ x y))
3
```

- A `let*` is functionally equivalent to a series of nested `let`s

- In both `let` and `let*`, initial values default to `nil`

Variables

- The `defparameter` operator introduces new global variables

```
CL-USER> (defparameter *glob* 99)
*GLOB*
```

- Accessible from everywhere
  - Except in expressions with local variable with the same name
  - To distinguish global variables, use the convention of starting and ending a global variable name with an asterisk

Variables

- The `defconstant` operator introduces new global constants

```
CL-USER> (defconstant limit (+ *glob* 1))
```

Assignment

- The `setf` operator assigns values to local and global variables

```
CL-USER> (setf *glob* 98)
98
CL-USER> (let ((n 10))
           (setf n 2) ;overwrites the 10
           n)
2
```

- When the first argument to `setf` is not the name of a local variable, it is taken to be a global variable (and is created, if necessary)

```
CL-USER> (setf x (list 'a 'b 'c))
(A B C)
```

Assignment

- The first argument to `setf` can be an expression
- In this case, the second argument is inserted in the place referred to by the first, i.e., the expression

```
CL-USER> (setf (car x) 'n)
N
CL-USER> x
(N B C)
```

Assignment

- You can give any (even) number of arguments to `setf`

```
CL-USER> (setf a b
               c d
               e f)
```

Functional Programming

- Writing programs that work by returning values, instead of modifying things

- It is the dominant paradigm in Lisp

```
CL-USER> (setf lst `(c a r a t))
(C A R A T)
CL-USER> (remove 'a lst)
(C R T)
CL-USER> lst
(C A R A T)
```

Removes all the `a` elements and returns a list

But `lst` is unchanged

Functional Programming

- To remove the element from the list, rather than just returning a list with the value removed:

```
CL-USER> (setf x (remove 'a x))
```

- Functional programming avoids `setf` and similar constructs
- Focus on returning values rather than the side-effects
- Functional programming facilitates interactive testing

Iteration

- The `do` macro is the fundamental iteration operator in Lisp

- Like `let`, `do` can create variables

- The first argument is a list of variable specifications of the form (*variable initial update*)

- Initially, each variable is set to the value of the *initial* expression

- On each iteration, each variable is set to the value of the *update* expression

Iteration

- The second argument is a list containing one or more expressions
- The first of these is used to test whether iteration should stop
- The remaining expressions in this list will be evaluated in order when iteration stops
  - The value of the last expression is returned as the value of the `do`

Iteration

- The remaining arguments comprise the body of the loop
- They are evaluated, in order, on each iteration
- One each iteration the variables are updated,
  then the termination test is evaluated,
  and then (if the test failed) the body is evaluated

Iteration

```
(defun show-squares (start end)
  (do ((i start (+ i 1)))
      ((> i end) 'done)
     (format t "~A ~A~%" i (* i i))))

CL-USER> (show-squares 2 5)
2 4
3 9
4 16
5 25
DONE
```

Iteration

- The `dolist` macro iterates through the elements of a list
- The first argument is a list of the form (*variable expression*)
- This is followed by a body of expressions
- The body is evaluated with *variable* bound to successive elements of the list returned by *expression*

Iteration

Here is a function that returns the length of a list

```
(defun our-length (1st)
  (let ((len 0))
    (dolist (obj 1st) ;for each obj in lst
      (setf len (+ len 1)))
    len))
```

Functions as Objects

- In Lisp, functions are regular objects, like symbols, strings, or lists
- If we give the name of a function to `function`, it will return the associated object
- `function` is a special operator so we don't have to quote the argument

```
CL-USER> (function +)
#<Compiled-Function + 17BA4E>
```

Functions as Objects

- Just as we can use ' as an abbreviation for `quote`, we can use #' as an abbreviation for function:

```
CL-USER> #'+
#<Compiled-Function + 17BA4E>
```

- This abbreviation is known as a sharp-quote

Functions as Objects

- We can pass functions as arguments
- The function `apply` takes a function as an argument, along with a list of arguments for that function, and returns the result of applying the function to the arguments

```
CL-USER> (apply #'+ '(1 2 3))
6
CL-USER> (+ 1 2 3)
6
```

Apply the + operator ...

... to this list of arguments

Functions as Objects

- It can be given any number of arguments, so long as the last argument is a list

- The function `apply` takes a function as an argument, along with a list of arguments for that function, and returns the result of applying the function to the arguments

```
CL-USER> (apply #'+ 1 2 '(3 4 5))
15
```

Functions as Objects

- The function `funcall` does the same thing but does not need the arguments to be packaged in a list

```
CL-USER> (funcall #'+ 1 2 3)
6
```

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`