Robot Programming with Lisp

3. Functional Programming: Functions, Lexical Scope and Closures

Gayane Kazhoyan

Institute for Artificial Intelligence
Universität Bremen

19th April, 2016
Outline

Background

Theory

Organizational

Gayane Kazhoyan
19th April, 2016

Robot Programming with Lisp
Functional Programming

Pure functional programming concepts include:

- no program state (e.g. no global variables);
Functional Programming

Pure functional programming concepts include:

- no program state (e.g. no global variables);
- referential transparency, i.e. a function called twice with same arguments always generates the same output;
Functional Programming

Pure functional programming concepts include:

- no program *state* (e.g. no global variables);
- *referential transparency*, i.e. a function called twice with same arguments always generates the same output;
- functions don’t have *side effects*;
**Functional Programming**

Pure functional programming concepts include:

- no program *state* (e.g. no global variables);
- *referential transparency*, i.e. a function called twice with same arguments always generates the same output;
- functions don’t have *side effects*;
- avoid mutable data, i.e. once created, data structure values don’t change (*immutable data*);
Functional Programming

Pure functional programming concepts include:

- no program state (e.g. no global variables);
- referential transparency, i.e. a function called twice with same arguments always generates the same output;
- functions don’t have side effects;
- avoid mutable data, i.e. once created, data structure values don’t change (immutable data);
- heavy usage of recursions, as opposed to iterative approaches;
Functional Programming

Pure functional programming concepts include:

- no program state (e.g. no global variables);
- referential transparency, i.e. a function called twice with same arguments always generates the same output;
- functions don’t have side effects;
- avoid mutable data, i.e. once created, data structure values don’t change (immutable data);
- heavy usage of recursions, as opposed to iterative approaches;
- functions as first class citizens, as a result, higher-order functions (simplest analogy: callbacks);
Functional Programming

Pure functional programming concepts include:

• no program state (e.g. no global variables);

• referential transparency, i.e. a function called twice with same arguments always generates the same output;

• functions don’t have side effects;

• avoid mutable data, i.e. once created, data structure values don’t change (immutable data);

• heavy usage of recursions, as opposed to iterative approaches;

• functions as first class citizens, as a result, higher-order functions (simplest analogy: callbacks);

• lazy evaluations, i.e. only execute a function call when its result is actually used;
Functional Programming

Pure functional programming concepts include:

- no program state (e.g. no global variables);
- referential transparency, i.e. a function called twice with same arguments always generates the same output;
- functions don’t have side effects;
- avoid mutable data, i.e. once created, data structure values don’t change (immutable data);
- heavy usage of recursions, as opposed to iterative approaches;
- functions as first class citizens, as a result, higher-order functions (simplest analogy: callbacks);
- lazy evaluations, i.e. only execute a function call when its result is actually used;
- usage of lists as a main data structure; ....
Popular Languages

- **Scheme**: 1975, latest release in 2013, introduced many core functional programming concepts that are widely accepted today.
Popular Languages

- **Scheme**: 1975, latest release in 2013, introduced many core functional programming concepts that are widely accepted today
- **Common Lisp**: 1984, latest release (SBCL) in 2016, successor of Scheme, possibly the most influential, general-purpose, widely-used Lisp dialect
Popular Languages

- **Scheme**: 1975, latest release in 2013, introduced many core functional programming concepts that are widely accepted today

- **Common Lisp**: 1984, latest release (SBCL) in 2016, successor of Scheme, possibly the most influential, general-purpose, widely-used Lisp dialect

- **Erlang**: 1986, latest release in 2016, focused on concurrency and distributed systems, supports hot patching, used within AWS
Popular Languages

- **Scheme**: 1975, latest release in 2013, introduced many core functional programming concepts that are widely accepted today
- **Common Lisp**: 1984, latest release (SBCL) in 2016, successor of Scheme, possibly the most influential, general-purpose, widely-used Lisp dialect
- **Erlang**: 1986, latest release in 2016, focused on concurrency and distributed systems, supports hot patching, used within AWS
- **Haskell**: 1990, latest release in 2010, purely functional, in contrast to all others in this list
Popular Languages

- **Scheme**: 1975, latest release in 2013, introduced many core functional programming concepts that are widely accepted today
- **Common Lisp**: 1984, latest release (SBCL) in 2016, successor of Scheme, possibly the most influential, general-purpose, widely-used Lisp dialect
- **Erlang**: 1986, latest release in 2016, focused on concurrency and distributed systems, supports hot patching, used within AWS
- **Haskell**: 1990, latest release in 2010, purely functional, in contrast to all others in this list
- **Racket**: 1994, latest release in 2016, focused on writing domain-specific programming languages
Popular Languages [2]

- **OCaml**: 1996, latest release in 2015, very high performance, static-typed, one of the first inherently object-oriented functional programming languages
Popular Languages [2]

- **OCaml**: 1996, latest release in 2015, very high performance, static-typed, one of the first inherently object-oriented functional programming languages

- **Scala**: 2003, latest release in 2016, compiled to JVM code, static-typed, object-oriented, Java-like syntax `{}`
Popular Languages [2]

- **OCaml**: 1996, latest release in 2015, very high performance, static-typed, one of the first inherently object-oriented functional programming languages
- **Scala**: 2003, latest release in 2016, compiled to JVM code, static-typed, object-oriented, Java-like syntax `{}`
- **Clojure**: 2007, latest release in 2016, compiled to JVM code and JavaScript, therefore mostly used in Web, seems to be fashionable in the programming subculture at the moment
Popular Languages [2]

- **OCaml**: 1996, latest release in 2015, very high performance, static-typed, one of the first inherently object-oriented functional programming languages

- **Scala**: 2003, latest release in 2016, compiled to JVM code, static-typed, object-oriented, Java-like syntax {}

- **Clojure**: 2007, latest release in 2016, compiled to JVM code and JavaScript, therefore mostly used in Web, seems to be fashionable in the programming subculture at the moment

- **Julia**: 2012, latest release in 2016, focused on high-performance numerical and scientific computing, means for distributed computation, strong FFI support, Python-like syntax
Popular Languages [2]

- **OCaml**: 1996, latest release in 2015, very high performance, static-typed, one of the first inherently object-oriented functional programming languages

- **Scala**: 2003, latest release in 2016, compiled to JVM code, static-typed, object-oriented, Java-like syntax {}

- **Clojure**: 2007, latest release in 2016, compiled to JVM code and JavaScript, therefore mostly used in Web, seems to be fashionable in the programming subculture at the moment

- **Julia**: 2012, latest release in 2016, focused on high-performance numerical and scientific computing, means for distributed computation, strong FFI support, Python-like syntax

**Conclusion**: functional programming becomes more and more popular.
Outline

Background

Theory

Organizational

Background

Theory

Organizational
Defining a Function

Signature

CL-USER> (defun my-cool-function-name (arg-1 arg-2 arg-3 arg-4)
    "This function combines its 4 input arguments into a list
    and returns it."
    (list arg-1 arg-2 arg-3 arg-4))

Optional Arguments

CL-USER> (defun optional-arguments (arg-1 arg-2 &optional arg-3 arg-4)
    (list arg-1 arg-2 arg-3 arg-4))
CL-USER> (optional-arguments 1 2 3 4)
(1 2 3 4)
CL-USER> (optional-arguments 1 2 3)
(1 2 3 NIL)
CL-USER> (optional-arguments 304)
invalid number of arguments: 1
Defining a Function [2]

Key Arguments

CL-USER> (defun specific-optional (arg-1 arg-2 &key arg-3 arg-4)
  "This function demonstrates how to pass a value to a specific optional argument."
  (list arg-1 arg-2 arg-3 arg-4))
SPECIFIC-OPTIONAL

CL-USER> (specific-optional 1 2 3 4)
unknown &KEY argument: 3

CL-USER> (specific-optional 1 2 :arg-4 4)
(1 2 NIL 4)
Defining a Function [3]

Unlimited Number of Arguments

```
CL-USER> (defun unlimited-args (arg-1 &rest args)
             (format t "Type of args is ~a.~%" (type-of args))
             (cons (list arg-1) args))

UNLIMITED-ARGS

CL-USER> (unlimited-args 1 2 3 4)
Type of args is CONS.
(1 2 3 4)

CL-USER> (unlimited-args 1)
Type of args is NULL.
(1)
```

Background

Gayane Kazhoyan
19th April, 2016

Theory

Organizational

Robot Programming with Lisp
Multiple Values

list vs. values

CL-USER> (defvar *some-list* (list 1 2 3))
*SOME-LIST*
CL-USER> *some-list*
(1 2 3)
CL-USER> (defvar *values?* (values 1 2 3))
*VALUES?*
CL-USER> *values?*
1
CL-USER> (values 1 2 3)
1
2
3
CL-USER> *
1
CL-USER>  //
(1 2 3)
Returning Multiple Values!

CL-USER> (defvar *db* '((Anna 1987) (Bob 1899) (Charlie 1980)))
  (defun name-and-birth-year (id)
     (values (first (nth (- id 1) *db*))
             (second (nth (- id 1) *db*)))))

NAME-AND-BIRTH-YEAR

CL-USER> (name-and-birth-year 2)
BOB
1899

CL-USER> (multiple-value-bind (name year) (name-and-birth-year 2)
         (format t "~a was born in ~a.~%" name year))
BOB was born in 1899.
NIL
Function Designators
Similar to C pointers or Java references

Designator of a Function

CL-USER> (describe '+)
COMMON-LISP:+
   [symbol]
+ names a special variable:
+ names a compiled function:
CL-USER> #'+
CL-USER> (symbol-function '+)
#<FUNCTION +>
CL-USER> (describe #'+)
#<FUNCTION +>
   [compiled function]
Lambda-list: (&REST NUMBERS)
Declared type: (FUNCTION (&REST NUMBER) (VALUES NUMBER &OPTIONAL))
Derived type: (FUNCTION (&REST T) (VALUES NUMBER &OPTIONAL))
Documentation: ...
Source file: SYS:SRC;CODE;NUMBERS.LISP

Background

Gayane Kazhoyan
19th April, 2016

Theory

Organizational

Robot Programming with Lisp
Higher-order Functions

Function as Argument

CL-USER> (funcall #'+ 1 2 3)
CL-USER> (apply #'+ '(1 2 3))
6
CL-USER> (defun transform-1 (num) (/ 1.0 num))
TRANSFORM-1
CL-USER> (defun transform-2 (num) (sqrt num))
TRANSFORM-2
CL-USER> (defun print-transformed (a-number a-function)
    (format t "~a transformed with ~a becomes ~a.~%"
            a-number a-function (funcall a-function a-number)))
PRINT-TRANSFORMED
CL-USER> (print-transformed 4 #'transform-1)
4 transformed with #<FUNCTION TRANSFORM-1> becomes 0.25.
CL-USER> (print-transformed 4 #'transform-2)
4 transformed with #<FUNCTION TRANSFORM-2> becomes 2.0.
CL-USER> (sort '(2 6 3 7 1 5) #'>)
(7 6 5 3 2 1)
Higher-order Functions [2]

Function as Return Value

```
CL-USER> (defun give-me-some-function ()
    (case (random 5)
      (0 #'+)
      (1 #'-)
      (2 #'*)
      (3 #'/)
      (4 #'values)))
GIVE-ME-SOME-FUNCTION

CL-USER> (give-me-some-function)
#<FUNCTION ->

CL-USER> (funcall (give-me-some-function) 10 5)
5

CL-USER> (funcall (give-me-some-function) 10 5)
2
```
Anonymous Functions

**lambda**

CL-USER> (sort '((1 2 3 4) (3 4) (6 3 6)) #'>)
The value (3 4) is not of type NUMBER.
CL-USER> (sort '((1 2 3 4) (3 4) (6 3 6))
   (lambda (x y)
     (> (length x) (length y))))
   ((1 2 3 4) (6 3 6) (3 4))
CL-USER> (sort '((1 2 3 4) (3 4) (6 3 6)) #'> :key #'car)
   ((6 3 6) (3 4) (1 2 3 4))

CL-USER> (defun random-generator-a-to-b (a b)
   (lambda () (+ (random (- b a)) a)))
RANDOM-GENERATOR-A-TO-B
CL-USER> (random-generator-a-to-b 5 10)
#<CLOSURE (LAMBDA () :IN RANDOM-GENERATOR-A-TO-B) {100D31F90B}>
CL-USER> (funcall (random-generator-a-to-b 5 10))
9

Background Theory Organizational

Gayane Kazhoyan
19th April, 2016
Robot Programming with Lisp
The let Environment

let

CL-USER> (let ((a 1)
              (b 2))
      (values a b))
1
2
CL-USER> (values a b)
The variable A is unbound.

CL-USER> (defvar some-var 'global)
     (let ((some-var 'outer))
      (let ((some-var 'inter))
        (format t "some-var inner: ~a~%" some-var))
        (format t "some-var outer: ~a~%" some-var))
        (format t "global-var: ~a~%" some-var)
?
The let Environment

```lisp
CL-USER> (let ((a 1)
                   (b 2))
           (values a b))
1
2
CL-USER> (values a b)
The variable A is unbound.

CL-USER> (defvar some-var 'global)
(let ((some-var 'outer))
    (let ((some-var 'inter))
        (format t "some-var inner: ~a~%" some-var))
    (format t "some-var outer: ~a~%" some-var))
    (format t "global-var: ~a~%" some-var)
some-var inner: INTER
some-var outer: OUTER
global-var: GLOBAL
```
The let Environment [2]

let*

CL-USER> (let ((a 4)
                (a^2 (expt a 2)))
            (values a a^2))

The variable A is unbound.

CL-USER> (let* ((a 4)
                 (a^2 (expt a 2)))
           (values a a^2))

4
16
Lexical Variables

In Lisp, non-global **variable values are**, when possible, **determined at compile time**. They are **bound lexically**, i.e. they are bound to the code they’re defined in, not to the run-time state of the program.

Riddle

```
CL-USER> (let* ((lexical-var 304)
                 (some-lambda (lambda () (+ lexical-var 100))))
           (setf lexical-var 4)
           (funcall some-lambda))
? 
```
Lexical Variables

In Lisp, non-global variable values are, when possible, determined at compile time. They are bound lexically, i.e. they are bound to the code they’re defined in, not to the run-time state of the program.

\[
\text{CL-USER> (let* ((lexical-var 304) }
\quad \text{(some-lambda (lambda () (+ lexical-var 100)))})
\quad \text{(setf lexical-var 4) }
\quad \text{(funcall some-lambda))}
\]

This is one single let block, therefore lexical-var is the same everywhere in the block.
Lexical Variables [2]

Lexical scope with \texttt{lambda} and \texttt{defun}

```
CL-USER> (defun return-x (x)
           (let ((x 304))
             x))
   (return-x 3)
```
Lexical Variables [2]

Lexical scope with `lambda` and `defun`

```
CL-USER> (defun return-x (x)
    (let ((x 304))
      x))
    (return-x 3)
```

`lambda`s and `defun`s create lexical local variables per default.
Lexical Variables [3]

More Examples

```
CL-USER> (let* ((lexical-var 304)
               (some-lambda (lambda () (+ lexical-var 100))))
               (setf lexical-var 4)
               (funcall some-lambda))
104
CL-USER> lexical-var
?
```
More Examples

CL-USER> (let* ((lexical-var 304)
             (some-lambda (lambda () (+ lexical-var 100))))
            (setf lexical-var 4)
            (funcall some-lambda))
104
CL-USER> lexical-var
; Evaluation aborted on #<UNBOUND-VARIABLE LEXICAL-VAR {100AA9C403}>.
CL-USER> (let ((another-var 304)
            (another-lambda (lambda () (+ another-var 100))))
            (setf another-var 4)
            (funcall another-lambda))
?
Lexical Variables [3]

More Examples

CL-USER> (let* ((lexical-var 304)
                 (some-lambda (lambda () (+ lexical-var 100))))
         (setf lexical-var 4)
         (funcall some-lambda))
104

CL-USER> lexical-var
; Evaluation aborted on #<UNBOUND-VARIABLE LEXICAL-VAR {100AA9C403}>

CL-USER> (let ((another-var 304)
                 (another-lambda (lambda () (+ another-var 100))))
         (setf another-var 4)
         (funcall another-lambda))
; caught WARNING:
;   undefined variable: ANOTHER-VAR
; Evaluation aborted on #<UNBOUND-VARIABLE ANOTHER-VAR {100AD51473}>

Background

Gayane Kazhoyan
19th April, 2016

Theory

Organizational

Robot Programming with Lisp
40
Lexical Variables [3]

More Examples

CL-USER> (let ((other-lambda (lambda () (+ other-var 100))))
  (setf other-var 4)
  (funcall other-lambda))

?
Lexical Variables [3]

More Examples

CL-USER> (let ((other-lambda (lambda () (+ other-var 100)))))
  (setf other-var 4)
  (funcall other-lambda))
; caught WARNING:
; undefined variable: OTHER-VAR
104
CL-USER> other-var
4
CL-USER> (describe 'other-var)
COMMON-LISP-USER::OTHER-VAR
    [symbol]
OTHER-VAR names an undefined variable:
    Value: 4
Lexical Variables [3]

More Examples

CL-USER> (let ((some-var 304))
  (defun some-fun () (+ some-var 100))
  (setf some-var 4)
  (funcall #'some-fun))
?

Background
Gayane Kazhoyan
19th April, 2016

Theory

Organizational
Robot Programming with Lisp
43
Lexical Variables [3]

More Examples

CL-USER> (let ((some-var 304))
  (defun some-fun () (+ some-var 100))
  (setf some-var 4)
  (funcall #'some-fun))

104

;; Alt-. on DEFUN brings you to "defboot.lisp"
(defmacro-mundanely defun (&environment env name args &body body)
  (multiple-value-bind (forms decls doc) (parse-body body)
    (let* ((lambda-guts `(,args ...))
            (lambda `(lambda ,@lambda-guts)) ...
    )
)
Closures

Counter

CL-USER> (defun increment-counter ()
       (let ((counter 0))
         (incf counter))
       (increment-counter)
       (increment-counter)

1

CL-USER> (defun increment-counter-closure ()
       (let ((counter 0))
         (lambda () (incf counter))))
INCREMENT-COUNTER-CLOSURE

CL-USER> (let ((function-object (increment-counter-closure)))
       (format t "counting: ~a ~a~%"
               (funcall function-object) (funcall function-object)))
counting: 1 2

Closure is a function that, in addition to its specific functionality, also encloses its lexical environment (environment as in, e.g., terminal environment variables).

Background Theory Organizational

Gayane Kazhoyan
19th April, 2016
Closures [2]

Counter Again

CL-USER> (defun increment-counter-lambda ()
  (let ((counter 0))
    (lambda (counter) (incf counter))))

INCREMENT-COUNTER-LAMBDA

CL-USER> (let ((function-object (increment-counter-lambda)))
  (format t "counter: ~a~%" (funcall function-object 0))
  (format t "once more: ~a~%" (funcall function-object 0)))

counter: 1
once more: 1

CL-USER> (let ((function-object (increment-counter-closure)))
  (format t "counter: ~a~%" (funcall function-object))
  (setf counter 0)
  (format t "counter: ~a~%" (funcall function-object)))

counter: 1

Encapsulation!

Background

Gayane Kazhoyan
19th April, 2016

Theory

Organizational

Robot Programming with Lisp

46
Lexical Variables [4]

Riddle #2

CL-USER> (let ((lex 'initial-value))

(defun return-lex ()
  lex)

(defun return-lex-arg (lex)
  (return-lex))

(format t "return-lex: ~a~%" (return-lex))
(format t "return-lex-arg: ~a~%" (return-lex-arg 'new-value))
(format t "return-lex again: ~a~%" (return-lex))

?
Lexical Variables [4]

Riddle #2

CL-USER> (let ((lex 'initial-value))
  (defun return-lex ()
    lex)
  (defun return-lex-arg (lex)
    (return-lex))
  (format t "return-lex: ~a~%
    (return-lex))
  (format t "return-lex-arg: ~a~%
    (return-lex-arg 'new-value))
  (format t "return-lex again: ~a~%
    (return-lex)))

; caught STYLE-WARNING:
; The variable LEX is defined but never used.
return-lex: INITIAL-VALUE
return-lex-arg: INITIAL-VALUE
return-lex again: INITIAL-VALUE

Background | Theory | Organizational
---|---|---
Gayane Kazhoyan | Robot Programming with Lisp
19th April, 2016 |
Dynamic Variables

Riddle #3

```lisp
CL-USER> (defvar dyn 'initial-value)
CL-USER> (defun return-dyn ()
    dyn)
CL-USER> (defun return-dyn-arg (dyn)
    (return-dyn))
CL-USER>
(format t "return-dyn: ~a~%"
    (return-dyn))
(format t "return-dyn-arg: ~a~%"
    (return-dyn-arg 'new-value))
(format t "return-dyn again: ~a~%"
    (return-dyn))

? 
```
Dynamic Variables

Riddle #3

CL-USER> (defvar dyn 'initial-value)
CL-USER> (defun return-dyn ()
   dyn)
CL-USER> (defun return-dyn-arg (dyn)
   (return-dyn))
CL-USER>
(format t "return-dyn: ~a~%"
   (return-dyn))
(format t "return-dyn-arg: ~a~%"
   (return-dyn-arg 'new-value))
(format t "return-dyn again: ~a~%"
   (return-dyn))
  return-dyn: INITIAL-VALUE
  return-dyn-arg: NEW-VALUE
  return-dyn again: INITIAL-VALUE

defvar and defparameter create dynamically-bound variables.
Currying

Back to Generators

```lisp
CL-USER> (let ((x^10-lambda (lambda (x) (expt x 10))))
  (dolist (elem '(2 3))
    (format t "~a^10 = ~a~%" elem (funcall x^10-lambda elem))))
2^10 = 1024
3^10 = 59049
;; The following only works with roslisp_repl. Otherwise do first:
;; (pushnew #p"/.../alexandria" asdf:*central-registry* :test #'equal)
CL-USER> (asdf:load-system :alexandria)
CL-USER> (dolist (elem '(2 3))
  (format t "~a^10 = ~a~%"
    elem (funcall (alexandria:curry #'expt 10) elem)))
2^10 = 100
3^10 = 1000
CL-USER> (dolist (elem '(2 3))
  (format t "~a^10 = ~a~%"
    elem (funcall (alexandria:rcurry #'expt 10) elem)))
2^10 = 1024
3^10 = 59049
```

Background Theory Organizational

Gayane Kazhoyan
19th April, 2016
Guidelines

• Don’t use global variables! Only for constants.
• If your function generates side-effects, name it correspondingly (either `foo!` which is preferred, or `foof` as in `setf`, or `nfoo` as in `nconc`)
• Use `Ctrl-Alt-\` on a selected region to fix indentation
• Try to keep the brackets all together:

This looks weird in Lisp

```
(if condition
  do-this
  do-that
)
```
Links

- Alexandria documentation:
  
  http://common-lisp.net/project/alexandria/draft/alexandria.html
Outline

Background

Theory

Organizational

Gayane Kazhoyan
19th April, 2016

Robot Programming with Lisp
Info Summary

• Assignment code: REPO/assignment_3/src/...
• Assignment points: 10 out of 50
• Assignment due: 26.04, Tuesday, 08:00 AM German time
• Next class: 26.04, 16:15
Thanks for your attention!