Robot Programming with Lisp

4. Functional Programming:
   Higher-order Functions, Map/Reduce, Lexical Scope

Gayane Kazhoyan

Institute for Artificial Intelligence
University of Bremen

8th of November, 2018
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.

- **Common Lisp**: 1984, latest release (SBCL) in 2018, successor of Scheme, possibly the most influential, general-purpose, widely-used Lisp dialect.

- **Erlang**: 1986, latest release in 2018, focused on concurrency and distributed systems, supports hot patching, used within AWS.

- **Haskell**: 1990, latest release in 2010 (new release planned in 2020), 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 2018, 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 2018, successor of Scheme, possibly the most influential, general-purpose, widely-used Lisp dialect.

- **Erlang**: 1986, latest release in 2018, focused on concurrency and distributed systems, supports hot patching, used within AWS.

  Background  Concepts  Organizational
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 2018, successor of Scheme, possibly the most influential, general-purpose, widely-used Lisp dialect
- **Erlang**: 1986, latest release in 2018, focused on concurrency and distributed systems, supports hot patching, used within AWS
- **Haskell**: 1990, latest release in 2010 (new release planned in 2020), 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 2018, successor of Scheme, possibly the most influential, general-purpose, widely-used Lisp dialect.

- **Erlang**: 1986, latest release in 2018, focused on concurrency and distributed systems, supports hot patching, used within AWS.

- **Haskell**: 1990, latest release in 2010 (new release planned in 2020), purely functional, in contrast to all others in this list.

Popular Languages [2]

- **OCaml**: 1996, latest release in 2018, very high performance, static-typed, one of the first inherently object-oriented functional programming languages

- **Scala**: 2003, latest release in 2018, compiled to JVM code, static-typed, object-oriented, Java-like syntax

- **Clojure**: 2007, latest release in 2017, 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 2018, 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.
Popular Languages [2]

- **OCaml**: 1996, latest release in 2018, very high performance, static-typed, one of the first inherently object-oriented functional programming languages.
- **Scala**: 2003, latest release in 2018, compiled to JVM code, static-typed, object-oriented, Java-like syntax {}
Popular Languages [2]

- **OCaml**: 1996, latest release in 2018, very high performance, static-typed, one of the first inherently object-oriented functional programming languages

- **Scala**: 2003, latest release in 2018, compiled to JVM code, static-typed, object-oriented, Java-like syntax `{}`

- **Clojure**: 2007, latest release in 2017, 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 2018, very high performance, static-typed, one of the first inherently object-oriented functional programming languages

- **Scala**: 2003, latest release in 2018, compiled to JVM code, static-typed, object-oriented, Java-like syntax {}

- **Clojure**: 2007, latest release in 2017, 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 2018, 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 2018, very high performance, static-typed, one of the first inherently object-oriented functional programming languages
- **Scala**: 2003, latest release in 2018, compiled to JVM code, static-typed, object-oriented, Java-like syntax {}
- **Clojure**: 2007, latest release in 2017, 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 2018, 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.
Contents

Background

Concepts

Functions Basics
Higher-order Functions
Anonymous Functions
Currying
Mapping and Reducing
Lexical Scope

Organizational

Gayane Kazhoyan
8th of November, 2018
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
```
Artificial Intelligence

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 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)
**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)
```
Multiple Values [2]

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
Contents

Background

Concepts

Functions Basics
Higher-order Functions
Anonymous Functions
Currying
Mapping and Reducing
Lexical Scope

Organizational
Higher-order Functions

Function as Argument

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

Background

Concepts

Functions Basics
Higher-order Functions
Anonymous Functions
Currying
Mapping and Reducing
Lexical Scope

Organizational
Anonymous Functions

### lambdas

```lisp
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)) #'> :key #'car)
((6 3 6) (3 4) (1 2 3 4))

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> (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
```
Contents

Background

Concepts

- Functions Basics
- Higher-order Functions
- Anonymous Functions
- Currying
- Mapping and Reducing
- Lexical Scope

Organizational
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
```
Contents

Background

Concepts
- Functions Basics
- Higher-order Functions
- Anonymous Functions
- Currying
- Mapping and Reducing
- Lexical Scope

Organizational
Mapping

Mapping in functional programming is the process of applying a function to all members of a list, returning a list of results.

Supported in most functional programming languages and, in addition

- C++ (STL)
- JavaScript 1.6+
- Matlab
- Java 8+
- PHP 4.0+
- Perl
- Python 1.0+
- Ruby
- Prolog
- C# 3.0+
- Mathematica
- Smalltalk, ...

In some of the languages listed the implementation is limited and not elegant.
Mapping [2]

mapcar is the standard mapping function in Common Lisp.

\[ \text{mapcar function list-1 \& rest more-lists} \Rightarrow \text{result-list} \]

Apply function to elements of list-1. Return list of function return values.

---

**mapcar**

```
CL-USER> (mapcar #'abs '(-2 6 -24 4.6 -0.2d0 -1/5))
(2 6 24 4.6 0.2d0 1/5)
CL-USER> (mapcar #'list '(1 2 3 4))
((1) (2) (3) (4))
CL-USER> (mapcar #'second '((1 2 3) (a b c) (10/3 20/3 30/3)))
?
CL-USER> (mapcar #'+ '(1 2 3 4 5) '(10 20 30 40))
?
CL-USER> (mapcar #'cons '(a b c) '(1 2 3))
?
CL-USER> (mapcar (lambda (x) (expt 10 x)) '(2 3 4))
?
```
Mapping [2]

`mapcar` is the standard mapping function in Common Lisp.

`mapcar function list-1 &rest more-lists ⇒ result-list`

Apply `function` to elements of `list-1`. Return list of `function` return values.

```
CL-USER> (mapcar #'abs '(-2 6 -24 4.6 -0.2d0 -1/5))
(2 6 24 4.6 0.2d0 1/5)
CL-USER> (mapcar #'list '(1 2 3 4))
((1) (2) (3) (4))
CL-USER> (mapcar #'second '((1 2 3) (a b c) (10/3 20/3 30/3)))
(2 B 20/3)
CL-USER> (mapcar #'+ '(1 2 3 4 5) '(10 20 30 40))
(11 22 33 44)
CL-USER> (mapcar #'cons '(a b c) '(1 2 3))
((A . 1) (B . 2) (C . 3))
CL-USER> (mapcar (lambda (x) (expt 10 x)) '(2 3 4))
(100 1000 10000)
```
Mapping [3]

`mapc` is mostly used for functions with side effects.

```
mapc function list-1 &rest more-lists ⇒ list-1
```

```
CL-USER> (mapc #'set '(*a* *b* *c*) '(1 2 3))
(*A* *B* *C*)
CL-USER> *c*
3
CL-USER> (mapc #'format '(t t) '("hello, " "world~\%"))
hello, world
(T T)
CL-USER> (mapc (alexandria:curry #'format t) '("hello, " "world~\%"))
hello, world
("hello~\%" "world~\%")
CL-USER> (mapc (alexandria:curry #'format t "~a ") '(1 2 3 4))
1 2 3 4
(1 2 3 4)
CL-USER> (let (temp)
  (mapc (lambda (x) (push x temp)) '(1 2 3))
temp)
```

Background Concepts Organizational
Mapping [4]

mapcan combines the results using \texttt{nconc} instead of \texttt{list}.

**mapcan function list-1 \& rest more-lists \Rightarrow concatenated-results**

If the results are not lists, the consequences are undefined.

\textbf{nconc vs list}

\begin{verbatim}
CL-USER> (list '(1 2) nil '(3 45) '(4 8) nil)
((1 2) NIL (3 45) (4 8) NIL)
CL-USER> (nconc '(1 2) nil '(3 45) '(4 8) nil)
(1 2 3 45 4 8)
CL-USER> (nconc '(1 2) nil 3 '(45) '(4 8) nil)
; Evaluation aborted on #<TYPE-ERROR expected-type: LIST datum: 1>.
CL-USER> (let ((first-list (list 1 2 3))
  (second-list (list 4 5)))
  (values (nconc first-list second-list)
    first-list
    second-list))
(1 2 3 4 5)
\end{verbatim}
Mapping [4]

`mapcan` combines the results using `nconc` instead of `list`.

`mapcan function list-1 &rest more-lists ⇒ concatenated-results`

If the results are not lists, the consequences are undefined.

```
(mapcar #'list '(1 2 3))
((1) (2) (3))
(mapcan #'list '(1 2 3))
(1 2 3)
(mapcan #'alexandria:iota '(1 2 3))
(0 0 1 0 1 2)
(mapcan (lambda (x)
            (when (numberp x)
                (list x)))
            '(4 n 1/3 ";")
(4 1/3)
```
Mapping [5]

maplist, mpl and mapcon operate on sublists of the input list.

maplist function list-1 &rest more-lists ⇒ result-list

maplist

CL-USER> (mapcar #'identity '(1 2 3))
(1 2 3)
CL-USER> (maplist #'identity '(1 2 3))
((1 2 3) (2 3) (3))
CL-USER> (maplist (lambda (x)
  (when (>= (length x) 2)
    (- (second x) (first x))))
  '(2 2 3 3 2 3 2 3 2 2 3))
  · · · · · · · · · · · · · · · · · · ·
  · · · · · · · · · · · · · · · · · · ·
  (0 1 0 0 -1 1 -1 1 -1 0 1 NIL)
  · · · · · · · · · · · · · · · · · · ·
CL-USER> (maplist (lambda (a-list) (apply #'* a-list)) '(4 3 2 1))
(24 6 2 1)
Mapping [5]

maplist, mapl and mapcon operate on sublists of the input list.

mapl function list-1 &rest more-lists ⇒ list-1

mapcon function list-1 &rest more-lists ⇒ concatenated-results

mapl

CL-USER> (let (temp)
  (mapl (lambda (x) (push x temp)) '(1 2 3))
  temp)
((3) (2 3) (1 2 3))

mapcon

CL-USER> (mapcon #'reverse '(4 3 2 1))
(1 2 3 4 1 2 3 1 2 1)
CL-USER> (mapcon #'identity '(1 2 3 4))
; Evaluation aborted on NIL.
Mapping [6]

`map` is a generalization of `mapcar` for `sequences` (lists and vectors).

`map` result-type function first-sequence &rest more-sequences ⇒ result

```lisp
(mapcar #'+ #(1 2 3) #(10 20 30))
The value #(1 2 3) is not of type LIST.
(map 'vector #'+ #(1 2 3) #(10 20 30))
#(11 22 33)
(map 'list #'+ '(1 2 3) '(10 20 30))
(11 22 33)
(map 'list #'identity '('#\h #\e #\l #\l #\o'))
('hello')
```
Reduction

**reduce** function sequence &key key from-end start end initial-value ⇒ result

Uses a binary operation, *function*, to combine the elements of *sequence*.

---

**reduce**

```
CL-USER> (reduce (lambda (x y) (list x y)) '(1 2 3 4))
((1 2) 3) 4)
CL-USER> (reduce (lambda (x y) (format t "~a ~a~%" x y)) '(1 2 3 4))
1 2
NIL 3
NIL 4
CL-USER> (reduce #'+ '()) ; ?
CL-USER> (reduce #'cons '(1 2 3 nil))
?
CL-USER> (reduce #'cons '(1 2 3) :from-end t :initial-value nil)
?
CL-USER> (reduce #'+ '((1 2) (3 4) (5 6))
    :key #'first :start 1 :initial-value -10)
?```
Reduction

**reduce** function sequence &key key from-end start end initial-value ⇒ result

Uses a binary operation, *function*, to combine the elements of *sequence*.

```
CL-USER> (reduce (lambda (x y) (list x y)) '(1 2 3 4))
((1 2) 3 4)
CL-USER> (reduce (lambda (x y) (format t "~a ~a~%" x y)) '(1 2 3 4))
1 2
NIL 3
NIL 4
CL-USER> (reduce #'+ '()) ; ?
CL-USER> (reduce #'cons '(1 2 3 nil))
((1 . 2) . 3))
CL-USER> (reduce #'cons '(1 2 3) :from-end t :initial-value nil)
(1 2 3)
CL-USER> (reduce #'+ '((1 2) (3 4) (5 6))
   :key #'first :start 1 :initial-value -10)
-2 ; = -10 + 3 + 5
```

Background | Concepts | Organizational
---|---|---
Gayane Kazhoyan
8th of November, 2018

Robot Programming with Lisp
MapReduce

Google’s *MapReduce* is a programming paradigm used mostly in huge databases for distributed processing. It was originally used for updating the index of the WWW in their search engine.

Currently supported by AWS, MongoDB, ...

Inspired by the map and reduce paradigms of functional programming.

https://en.wikipedia.org/wiki/MapReduce
MapReduce [2]

Example

**Task:** calculate at which time interval the number of travelers on the tram is the highest (intervals are “early morning”, “late morning”, ...)

**Database:** per interval hourly entries on number of travelers (e.g. db_early_morning: 6:00 → Tram6 → 100, 7:00 → Tram8 → 120)

**Map step:** per DB, go through tram lines and sum up travelers:

- *DB1 early morning:* (Tram6 → 2000) (Tram8 → 1000) ...
- *DB6 late night:* (Tram6 → 200) (Tram4 → 500) ...

**Reduce:** calculate maximum of all databases for each tram line:
Tram6 → 3000 (late morning)
Tram8 → 1300 (early evening)

...
Contents

Background

Concepts
  - Functions Basics
  - Higher-order Functions
  - Anonymous Functions
  - Currying
  - Mapping and Reducing
  - Lexical Scope

Organizational

Gayane Kazhoyan
8th of November, 2018
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)
)`
The let Environment

```
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

```lisp
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.

Riddle

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

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

Lexical scope with `lambda` and `defun`

```lisp
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)
304
```

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

More Examples

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

Background Concepts Organizational

Gayane Kazhoyan
8th of November, 2018

Robot Programming with Lisp
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))
?
Lexical Variables [3]

More Examples

```lisp
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

Concepts

Organizational
Lexical Variables [3]

More Examples

```lisp
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))

?
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))
            (lambda `(lambda ,@lambda-guts))
            ...
            ...))
    ...)
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)))

Background Concepts Organizational

Gayane Kazhoyan
8th of November, 2018

Robot Programming with Lisp

63
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
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))
?
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))

return-dyn: INITIAL-VALUE
return-dyn-arg: NEW-VALUE
return-dyn again: INITIAL-VALUE
```

defvar and defparameter create dynamically-bound variables.

Background

Concepts

Organizational

Gayane Kazhoyan

8th of November, 2018
Local Function Definitions

\textit{flet}

\begin{verbatim}
CL-USER> (defun some-pseudo-code ()
          (flet ((do-something (arg-1)
                  (format t "doing something ~a now...~\%" arg-1)))
                (format t "hello.~\%")
                (do-something "nice")
                (format t "hello once again.~\%")
                (do-something "evil")))
SOME-PSEUDO-CODE
CL-USER> (some-pseudo-code)
hello.
doing something nice now...
hello once again.
doing something evil now...
NIL
CL-USER> (do-something)
; Evaluation aborted on #<UNDEFINED-FUNCTION DO-SOMETHING {101C7A9213}>
\end{verbatim}
Local Function Definitions [2]

flet, labels

CL-USER> (let* ((lexical-var 304)
    (some-lambda (lambda () (+ lexical-var 100))))
    (let ((lexical-var 4))
        (funcall some-lambda)))
; ?
CL-USER> (let ((lexical-var 304))
    (flet ((some-function () (+ lexical-var 100)))
        (let ((lexical-var 4))
            (some-function))))
; ?
Local Function Definitions [2]

flet, labels

CL-USER> (let* ((lexical-var 304)
    (some-lambda (lambda () (+ lexical-var 100))))
    (let ((lexical-var 4))
      (funcall some-lambda)))
404

CL-USER> (let ((lexical-var 304))
    (flet ((some-function () (+ lexical-var 100)))
      (let ((lexical-var 4))
        (some-function))))
404

CL-USER> (labels ((first-fun () (format t "inside FIRST~%"))
    (second-fun ()
      (format t "inside SECOND~%")
      (first-fun))
    (second-fun))

inside SECOND
inside FIRST
Contents

Background

Concepts
  Functions Basics
  Higher-order Functions
  Anonymous Functions
  Currying
  Mapping and Reducing
  Lexical Scope

Organizational

Gayane Kazhoyan
8th of November, 2018

Robot Programming with Lisp
Guidelines

• Avoid global variables! Use 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:

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

This looks weird in Lisp
• Alexandria documentation:
  http://common-lisp.net/project/alexandria/draft/alexandria.html
Info Summary

- **Assignment code**: REPO/assignment_4/src/...
- **Assignment points**: 10 points
- **Assignment due**: 14.11, Wednesday, 23:59 German time
- **Next class**: 15.11, 14:15
Thanks for your attention!