Common Lisp and Introduction to Functional Programming
Lecture 3: Common Lisp Functions and Variables

Yuri Zhykin

Feb 17, 2021

Function Definitions

- Functions in Common Lisp are defined with `defun` form:

```
(defun <function-name> <parameter-list> <docstring>? <body>)
```

- Example:

```
(defun greet (name formally?)
  "Print a greeting for the given `name`, either formal
or informal one, depending on the value of `formally?`."
  (if formally?
      (format t "Good morning, ~a.~%" name)
      (format t "What's up, ~a!~%" name)))

(greet "Mr. Holmes" t)
;; Good morning, Mr. Holmes.
(greet "Mike" nil)
;; What's up, Mike!
```

λ-lists 1/4

- Parameter lists in Common Lisp function definitions are called λ-lists (or lambda-lists).
- Regular symbols in λ-lists represent required parameters - missing required arguments cause an error to be signaled:

```
(defun f (x y z) ; x, y, z are parameters
  (* x (+ y z)))

(f 1 2) ; 1, 2 are arguments for x, y; argument for z is missing
;; Error: f got 2 args, wanted 3 args.
```

λ-lists 2/4

- **Optional parameters** are parameters that have default value and do not have to be provided.
- Optional parameters are positional: in order to provide an optional argument, all previous ones have to be included.
- Optional parameters are separated from the required ones with the special symbol `&optional`:

```
(defun greet (name &optional (formally? t) (time-of-day "morning"))
  (if formally?
      (format t "Good ~a, ~a.~%" time-of-day name)
      (format t "What's up, ~a!~%" name)))

(greet "Mr. Holmes")
;; Good morning, Mr. Holmes.
(greet "Mr. Cumberbatch" t "evening")
;; Good evening, Mr. Cumberbatch.
(greet "Mike" nil)
;; What's up, Mike!
```

λ-lists 3/4

- **Keyword parameters** are optional parameters that are specified in pairs (*name*, *value*).
- Keyword parameters can be specified out of order and selectively.
- Common Lisp uses special type of symbols called "keywords" to specify names for the optional parameters.
- Keyword parameters are separated from the others in the λ-list with the `&key` symbol:

```
(defun greet (name &key (formally? t) (time-of-day "morning"))
  ...)

(greet "Mr. Holmes" :time-of-day "afternoon")
;; Good afternoon, Mr. Holmes.
(greet "Mike" :formally? nil)
;; What's up, Mike!
```

λ-lists 4/4

- **Rest-parameter** declaration in a λ-list allows to collect a variable number of arguments into a list.
- Rest-argument includes any optional and keyword arguments that were supplied to the function.
- Rest-argument can be declared with the `&rest` symbol:

```
(defun f (zero &rest args &key key1 key2 &allow-other-keys)
  (format t "zero: ~a, key1: ~a, key2: ~a~%" zero key1 key2)
  (format t "rest: ~a~%" args))

(f 0 :key1 1 :key2 2 :key3 3 :key4 4)
;; zero: 0, key1: 1, key2: 2
;; rest: (:key1 1 :key2 2 :key3 3 :key4 4)
```

- Allowed set of keyword arguments is limited by default, and must be relaxed with the `&allow-other-keys` symbol.

λ-expressions (a.k.a. Anonymous Function)

- λ-**expressions** are *function literals* - then denote the functions without giving them names.
- Just regular function definitions, λ-expressions have two key components - λ-list and body. Unlike regular function definitions, λ-expressions do not have names.
- Compare:

```
CL-USER> (defun f (x) (* x 2))
f
CL-USER> (f 5)
10
```

and

```
CL-USER> (lambda (x) (* x 2))
#<Interpreted Function (unnamed) @ #x10008c1dbc2>
CL-USER> ((lambda (x) (* x 2)) 5)
10
```

Functions as First-class Values 1/4

- In programming languages, **first-class values** are values that can be *passed as arguments to functions* and *returned from functions as results*.
- Functions are fundamental mathematical objects, so in order to be "functional", programming language must consider functions *first-class values*.
- Because Common Lisp is a Lisp-2, it needs a special operation to call functions from namespace 1:

```
CL-USER> (let ((doubler (lambda (x) (* x 2))))
           (doubler 5))
;; Error: attempt to call `doubler' which is an undefined function.
CL-USER> (let ((doubler (lambda (x) (* x 2))))
           (funcall doubler 5))
10
```

Functions as First-class Values 2/4

- `funcall` applies the function to the arguments as if its λ-list was the same as of the function being applied:

```
CL-USER> (funcall #'+ 1 2 3 4)
10
```

- Sometimes it is more convenient to apply the function to the arguments collected into a list, which can be done with `apply`:

```
CL-USER> (apply #'+ '(1 2 3 4))
10
```

Functions as First-class Values 3/4

- Passing functions as arguments:

```
CL-USER> (defun twice (f x)
           (funcall f (funcall f x)))
twice
CL-USER> (twice (lamdba (x) (* x x)) 5)
125
```

- Returning functions as results:

```
CL-USER> (defun linear (a b)
           (lambda (x) (+ (* a x) b)))
linear
CL-USER> (let ((2x+5 (linear 2 5)))
           (funcall 2x+5 5))
15
```

Functions as First Class Values 4/4

- Finally, having functions as first-class values allows to compose them to build more complex functions:

```
(defun timed (f)
  (lambda (&rest args)
    (let* ((start-time (get-universal-time))
           (result (apply f args))
           (end-time (get-universal-time)))
      (format t "Elapsed time: ~a~%" (- end-time start-time))
      result)))

(defun parser (string)
  ...)

(funcall #'parser "<some string to be parsed>")

(funcall (timed #'parser) "<some string to be parsed>")
```

Useful Resources

- Common Lisp HyperSpec - publicly available alternative to ANSI Common Lisp standard
  - http://www.lispworks.com/documentation/HyperSpec/Front/index.htm
- Design Patterns in Dynamic Languages - Peter Norvig, 1998
  - https://norvig.com/design-patterns

The End

Thank you!