

Common Lisp and Introduction to Functional Programming

Lecture 4: State and Control Flow

Yuri Zhykin

Feb 17, 2021

Program State

- **Program's state** is the contents of all memory locations referenced by **variables**.
- **Variable** is a pair consisting of a storage location and a **symbolic name** that can be used to retrieve information stored in that storage location.

Variable Scope

- **Scope** refers to the textual region of the program, where references to some entity may occur.
- Common Lisp distinguishes the following:
 - **lexical scope** - references to the entity may only occur in the part of the program that is lexically (textually) contained within the construct that establishes it,
 - **indefinite scope** - references may occur anywhere in the program,

Variable Extent

- **Extent** refers to the interval of time during which the references to some entity may occur.
- Common Lisp distinguishes the following:
 - **dynamic extent** - references may only occur between the time control flow enters the construct that establishes the extent and exits it,
 - **indefinite extent** - entity continues to exist while there is still a possibility of reference.

Variables in Common Lisp 1/2

- Associations between the symbolic names of the variables and their values are called **bindings**.
- Variables can be **bound**, meaning there is a binding for that variable at a current point in the program, and **free**, meaning there is no binding.
- Variable bindings are stored in objects called **environments**, which can be nested:

```
;; Environment 0:
(let ((a 5)
      (b 10))
  ...
  ;; Environment 1:
  (let ((c (+ a b))
        (d (* a b)))
    ...
    (list a b c d))) ;;; Lookup in Env. 1, then Env. 0
```

Variables in Common Lisp 2/2

- Variables can be set with the `setf` form:

```
CL-USER> (let ((a 0))  
           (setf a 1)  
           a)  
1
```

- In the body of a function definition, variables representing the parameters are considered bound,

```
(defun f (a b)  
  (* a b))    ;; a and b are bound to the abstract "arguments"  
  
(f 1 2)  
;; The same as (almost, because in general variable  
;; renaming is needed):  
;; (let ((a 1)  
;;       (b 2))  
;;   (* a b))
```

Lexically-scoped Variables 1/2

- Simplest Common Lisp construct to create lexical bindings is a `let`-form:

```
CL-USER> (let ((a 5)
                (b 6))
           (* a b))
```

30

```
CL-USER> (* a b)
```

;; Error: Attempt to take the value of the unbound variable `a'.

- Here, variables `a` and `b` can only be accessed from within the body of the `let`-form.
- Scope of variables `a` and `b` is the lexical (textual) region `"(* a b)"`.
- Extent of variables `a` and `b` is the period while the form `(* a b)` executes.

Lexically-scoped Variables 2/2

- Lexical variables are invisible outside the construct that defines their scope:

```
CL-USER> (define f ()  
           (+ a b))
```

```
CL-USER> (let ((a 5)  
               (b 6))  
          (f))
```

;; Error: Attempt to take the value of the unbound variable `a'.

Dynamically-scoped Variables

- “**Dynamic scope**” is a hacky way of saying “indefinite scope and dynamic extent”.
- Common Lisp allows to declare variables to be “dynamically scoped”:

```
(defvar *coeff* 10) ; "special" (dynamically scoped) variable
```

```
(defun f (x) (* x *coeff*))
```

```
CL-USER> (f 5)
```

```
50
```

```
CL-USER> (let ((*coeff* 1000))  
           (f 5))
```

```
5000
```

```
CL-USER> *coefficient*
```

```
10
```

- Note that `let` can also create dynamic bindings.

Practical Example of Dynamically-scoped Variables

- Dynamic variables allow to implicitly parameterize code without having to pass around lots of arguments:

```
(defun print-numbers (n1 n2 n3)
  (print n1)
  (print n2)
  (print n3))
```

```
(print-numbers 10 11 12)
;; 10
;; 11
;; 12
```

```
(let ((*print-base* 16))
  (print-numbers 10 11 12))
;; a
;; b
;; c
```

Control Flow 1/4: sequence of forms

- Multiple Common Lisp forms can be executed in a sequence:

```
(progn
  <form1>
  <form2>
  ...
  <formN>)
```

- The value of the `progn` form is the value of the last form:

```
(progn
  (print "This will return 5.")
  5)
```

- The order can be switched with `prog1`:

```
(prog1 5
  (print "This will return 5."))
```

- A lot of Lisp forms that have a body, have an implicit `progn` around the body.

Control Flow 2/4: conditionals

- `if`-expression has the following form:

```
(if <condition>
   <true-branch-form>
   <false-branch-form>)
```

- There is a single-branch conditional:

```
(when <condition>
  <true-branch-body>
  ...)
;; (if <condition>
;;   (progn <true-branch-body>
;;         ...))
;;   nil
```

and a shortcut

```
(unless <condition>   ;; same as (when (not <condition>) ...)
  <false-branch-body>
  ...)
```

Control Flow 3/4: multiple values

- Common Lisp allows to return multiple values as a result of evaluating a form:

```
CL-USER> (values 1 2 3)
1
2
3
CL-USER> (multiple-value-bind (a b c) (values 1 2 3)
           (list 1 2 3))
(1 2 3)
CL-USER> (let ((a (values 1 2 3)))
           a)
1
```

- Used when the rest of the values are less significant than the main one.
- Unlike Python's `return` `a`, `b`, Common Lisp's `values` form does not pack/unpack values into a transient data structure.

Control Flow 4/4: simple iteration

- Common Lisp provides simple constructions to iterate over a list and repeat a body several times:

```
CL-USER> (dolist (element '(1 2 3))  
           (print element))
```

```
1  
2  
3
```

```
nil
```

```
CL-USER> (dotimes (i 3)  
           (print i))
```

```
0  
1  
2
```

```
nil
```

- There are also more complex `do` and `loop` forms.

Lexical Closures 1/2

- **Closure** is a pair consisting of a first-class function and an environment.
- Closure's **environment** maps the free variables in the function to the values they were bound to when the closure was created.
- Closure, unlike a regular function, can access the bindings in the environment where they were created.

```
(defun make-counter ()  
  (let ((c 0))  
    (lambda ()  
      (prog1 c  
        (incf c))))) ;; same as (setf c (+ c 1))
```

```
(let ((counter (make-counter)))  
  (funcall counter) ;; 0  
  (funcall counter) ;; 1
```

Lexical Closures 2/2

- Practical example of using lexical closures - **memoization**

```
(let ((cache (make-hash-table)))
```

```
  (defun %fib (n)
```

```
    (if (= n 0)
```

```
        0
```

```
        (if (= n 1)
```

```
            1
```

```
            (+ (%fib (- n 1)) (%fib (- n 2))))))
```

```
(defun fib (n)
```

```
  (multiple-value-bind (value exists?) (gethash n cache)
```

```
    (if exists?
```

```
        value
```

```
        (let ((value (%fib n)))
```

```
          (setf (gethash n cache) value)
```

```
          value))))))
```


Lexical Closures 2/2

- Practical example of using lexical closures - **memoization**

```
(let ((cache (make-hash-table)))
```

```
  (defun %fib (n)
```

```
    (if (= n 0)
```

```
        0
```

```
        (if (= n 1)
```

```
            1
```

```
            (+ (fib (- n 1)) (fib (- n 2))))))
```

```
(defun fib (n)
```

```
  (multiple-value-bind (value exists?) (gethash n cache)
```

```
    (if exists?
```

```
        value
```

```
        (let ((value (%fib n)))
```

```
          (setf (gethash n cache) value)
```

```
          value))))))
```

- Common Lisp HyperSpec - publicly available alternative to ANSI Common Lisp standard
 - <http://www.lispworks.com/documentation/HyperSpec/Front/index.htm>

The End

Thank you!