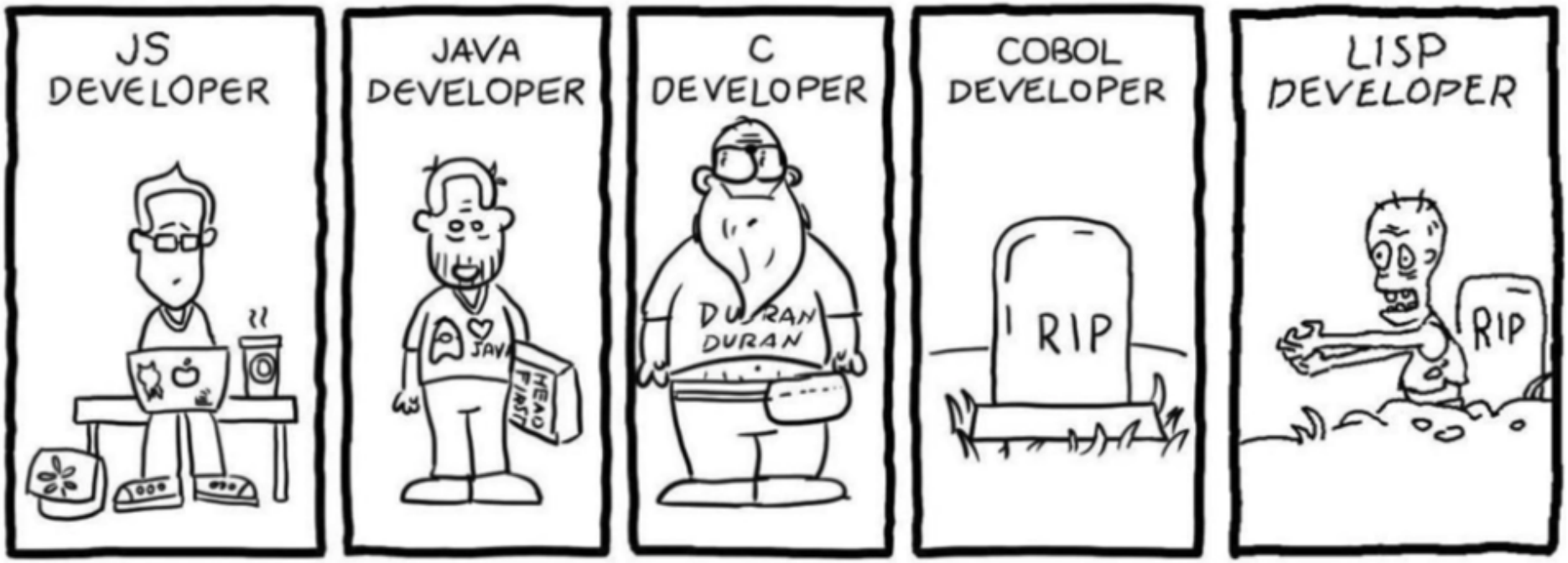
Week 01 - Lecture 2 Slides

Dr. Yeganeh Bahoo

Created: 2023-09-06 Wed 12:07

Table of Contents

Lecture 2: A crash course on Lisp (cont.)



Learning objectives:

By the end of this lecture you should be able to:

- Describe and write Lisp expressions that contain special forms
- Create programmer-defined functions
- Describe how functions are evaluated when applied to arguments

Lisp syntax and evaluation: special forms

- In the previous lecture we learned about the simplest lisp expressions (aka *forms*):
  - constants
  - symbols
  - function applications
- How about ifs, loops, programmer-defined functions, etc.?

- There are 25 "special forms" in Lisp!
- What makes them "special"?
  - Each special form follows their own evaluation rule.

Let's first learn the five most used special forms.

The most common special forms

- $(\text{QUOTE } v)$, also written as $'v$, where v is any lisp form. It means: return v as data, without evaluating it. (See HyperSpec doc)

```
(quote (+ 2 3))  ;==>   (+ 2 3)
'(+ 2 3)  ;==>  (+ 2 3)
(+ 2 3)  ;==>  5     ; Notice here that the form (+ 2 3) is not quoted
```

- $(AND\ a_1 \cdots a_n)$ and $(OR\ a_1 \cdots a_n)$: the forms a_i are evaluated from left to right
  - AND: As soon as any form a_i evaluates to nil, AND returns `NIL` without evaluating the remaining forms. If all forms but the last evaluate to true values, AND returns the result produced by evaluating the last form.
  - OR: As soon as any form a_i evaluates to true, OR returns a_i's value without evaluating the remaining forms. If all forms but the last evaluate to `NIL` values, OR returns the result produced by evaluating the last form.
- The following forms are interpreted as **false** in a predicate test: `()`, `'()`, `'NIL`, `NIL`
- Any value other than NIL or () is interpreted as **true**.

```
(and 1 2 3) ;==> 3
(and 0 1 nil (+ 3 4) 3)  ;==>  nil
(and 0 1 2 (* 3 3))  ;==> 9
(or 10 (* 3 3) 2 1)  ;==> 10
(or (> 5 6) (< 3 4) (= 1 2))  ;==>  T
```

Special forms (cont.): IF

$$(\text{IF}\ \ testForm\ \ thenForm\ \ [elseForm])$$

Note: in the meta-notation above, $[abc]$ means: abc is optional.

- Is a special form for conditional branching. Let's read its description and see examples at HyperSpec.

```
(if nil 4 (* 3 2))  ;==>  6

(if (and 0 1 nil (+ 3 4) 3)   ; Notice: indentation enhances readability
    3
    10)            ;==>  10

(if (or 0 (* 3 3) 2 1)
    (+ 5 5)
    (* 10 10))     ;==> 10
```

Homework: explore two other lisp conditional expressions: WHEN and UNLESS

Special forms (cont.): LAMBDA

- Used for creating a nameless function
- $(\text{LAMBDA } (x_1 \cdots x_n)\ e)$, where each x_i is a distinct symbol and e is a form. It evaluates to the function of n arguments which, when given arguments $v_1, \cdots v_n$, returns as its value the result of evaluating e in a context where the variable x_i is bound to v_i.

```
(lambda (x) (* x x))  ;==>  #<FUNCTION (LAMBDA (X)) {535F802B}>  - this is a special form


((lambda (x) (* x x)) 4)  ;==>  16               - this is a function application form


((lambda (x y) (+ (* x y) x)) 3 1)  ;==>  6      - this is also function application form
```

Special forms (cont.): LET

$$(\text{LET } \underbrace{((x_1\ e_1) \cdots (x_n\ e_n))}_{Bindings}\ e)$$

- first, all e_i are evaluated at once and their values bound, in parallel, to their respective x_i
- then e is evaluated

This is the usual way Lisp programmers give variables temporary local bindings (see HyperSpec doc)

```
(let ((x 4)) (* x x))  ;==>  16

(let ((x  3)                  ; Notice: indentation enhances readability
      (y 1))
   (+ (* x y) x))  ;==>  6

(let ((x 3)
      (y (* x x)))   <=== ERROR!!! 3 has not been bound to x yet!
  (+ x y))
```

Special forms (cont.): LET* and RUTILS:WITH

$$(\text{LET*}\ ((x_1\ e_1)\cdots(x_n\ e_n))\ e)$$

$$(\text{RUTILS:WITH}\ ((x_1\ e_1)\cdots(x_n\ e_n))\ e)$$

Unlike LET, in LET* (and in WITH) e_1 is evaluated and its value bound do x_1, then e_2 is evaluated and its value bound to x_2, and so on.

In a LET, all e_i are evaluated at once and their values bound, in parallel, to their respective x_i.

```
CL-USER> (let* ((a 2)         ; Notice: indentation enhances readability
                (b 4)
                (c -1)
                (delta (- (* b b) (* 4 a c)))
                (res (sqrt delta)))
           (/ (+ (- b) res) (* 2 a)))
0.22474492
```

- In the textbook, you may see RTL:WITH instead of LET*.
- WITH is a utility macro defined in the RUTILS package available in our lab machines.
- It offers more flexibility than LET* for creating bindings with elaborate data structures.

Exercise

What will the evaluation of each form return?

```
10  ;==>  ?

(+ 5 3 4)  ;==>  ?

(- 9 1)  ;==>  ?

(+ (* 2 4) (- 4 6))  ;==>  ?

(lambda (x y) (+ x y))  ;==>  ?

((lambda (x y) (+ x y)) (+ 2 1) 4)  ;==>  ?

(let ((a 4)
      (b 5)
      (c 30))
  (+ (* a b) c))  ;==>  ?
```

Solution

```
10  ;==>  10

(+ 5 3 4)  ;==>  12

(- 9 1)  ;==>  8

(+ (* 2 4) (- 4 6))  ;==>  6

(lambda (x y) (+ x y))  ;==>  #<FUNCTION (LAMBDA (X Y)) {2276001B}>

((lambda (x y) (+ x y)) (+ 2 1) 4)  ;==>  7

(let ((a 4)
      (b 5)
      (c 30))
  (+ (* a b) c))  ;==>  50
```

The most common special forms (cont.)

So far, we have identified in Lisp some elements that must appear in many programs:

- Numbers and arithmetic operations are primitive data and procedures

- Nesting of forms provides a means of combining operations
- LET expressions associate names with values providing a limited means of abstraction

Procedure definitions provide us a much more powerful abstraction technique.

The DEFUN special form

Syntax: $(\text{DEFUN } functionName\ (v_1 \cdots v_n) \underbrace{form^*}_{\text{Function body}})$, where:

- v_i are the parameters of the function
- $form^*$ means zero or more forms (see HyperSpec doc)

The DEFUN special form is for DEfining a FUNction.

```
CL-USER> (defun average (x y)
           "Returns the average of x and y"
            (/ (+ x y) 2.0))
```

- The DEFUN operator creates in memory a function/procedure indentified by its name and defined by its body.

```
(average (* 2 2) (/ 10 2)) ;==>  4.5
(describe 'average)   ;==>

AVERAGE names a compiled function:
...
Documentation:
  Returns the average of x and y
Source form:
  (LAMBDA (X Y)
    "Returns the average of x and y"
    (BLOCK AVERAGE (/ (+ X Y) 2.0)))
```

Example

Given that the built-in function ODDP tests if its argument is odd,

```
(oddp 3)  ;==>  T
```

```
(oddp 4)  ;==> NIL
```

write a function MAKE-EVEN that makes an odd number even by adding one to it. If the input to MAKE-EVEN is already even, it should be returned unchanged. For example:

```
(make-even 4)  ;==>  4

(make-even 5)  ;==>  6
```

Good styles	Bad style

`(defun make-even (x)`
`  (if (oddp x) (+ x 1)`
`  x))`
`;; Alternative style`
`(defun make-even (x)`
`  (if (oddp x)`
`  (+ x 1)`
`  x))`
	`(defun make-even (x)`
	`  (if (oddp x)`
	`  (+ x 1)`
	`  x))`

For more information abour Lisp programming style, read the Formatting section of this web page.

Quiz: defining functions

https://bit.ly/3Z3OAhT

Exercise

Write definitions for the following functions:

- `CUBE`: multiplies its input by itself three times.
- `ADD1`: adds one to its input
- `ADD2`: adds two to its input; it uses `add1` as a helper
- `MAX2`: returns the max if its two arguments, both are numbers
- `ONEMOREP`: returns `T` if its first input is exactly one greater than its second input; it uses `add1` as a helper

Solution

```
(defun cube (x)
  (* x x x))

(defun add1 (x)
  (+ x 1))

(defun add2 (x)
  (add1 (add1 x)))

(defun max2 (x y)
  (if (> x y) x y))

(defun onemorep (x y)
  (= x (add1 y)))
```