Writing More Flexible Functions

WIlensky Chapter 12

Parameter Designators

- Common Lisp has a variety of ways of handling parameters
  - Positional parameters
    - Standard method
    - Use of &optional
    - Use of &rest
  - Keyword parameters
  - Combinations of the above

Positional Parameters

- ◊ Positional parameters
  - » **Arguments passed by position in the argument sequence**
  - » **Standard method in most programming laguages –Java, C, Pascal, Turing, etc**
  - » **For example**

```
( defun  func  ( x  y  z )  ( .... )
( func  a  b  c )
```

> **Associate in pairs < a, x >, < b, y >, < c, z > by position first, second, third, etc.**

© Gunnar Gotshalks

Positional Parameters – &OPTIONAL

- **&optional** provides a means of adding a varying list of a small number of parameters
  - » **The programmer does not have to specify all the parameters**

- For example

```
(defun func (a  b  &optional x  (y  yDefault)
                                (z  zDefault  zPassed) ) ... )
```

> **Potential calling sequences**

```
(func a1 a2 )          – associate with a, b
(func a1 a2 a3)        – associate with a, b, x
(func a1 a2 a3 a4)     – associate with a, b, x, y
(func a1 a2 a3 a4 a5 ) – associate with a, b, x, y, z
```

- Note how “position” associates argument with parameter

&OPTIONAL details

- Consider the 3 optional parameters x, y and z are set

```
(defun func (a  b  &optional x  (y  yDefault)
                                  (z  zDefault  zPassed) ) ... )
```

- Can set default values for optional parameters
  - **If an argument is not given then the value of x is nil, of y is yDefault and of z is zDefault**

&OPTIONAL details – 2

- For example

```
(defun func (a  b  &optional x  (y  5)
                                   (z  ‘(1 2)  zPassed) ) ... )
```

  - Then calling the function with

» (func 1 2)

  - gives 1 -> a   2 -> b   nil -> x   5 -> y   (1 2) -> z

» (func 1 2 3)

  - gives 1 -> a   2 -> b   3 -> x   5 -> y   (1 2) -> z

» (func 1 2 3 4)

  - gives 1 -> a   2 -> b   3 -> x   4 -> y   (1 2) -> z

» (func 1 2 3 4 5)

  - gives 1 -> a   2 -> b   3 -> x   4 -> y   5 -> z

&OPTIONAL details - 3

◊ What about **zPassed** in the following

```
(defun func (a  b  &optional x  (y  5)
                                          (z  ‘(1 2)  zPassed) ) ... )
```

◊ It is used to indicate, within the function body, whether or not **z** was passed as a parameter

◊ For example the body could be

```
( cond  ( zPassed  ( print  “z was passed” ) )
        ( t  ( print  “z was not passed” ) ) )
```

◊ Then

```
( func  1  2  3 )          outputs    “z was not passed”
( func  1  2  3  4  5 )    outputs    “z was passed”
```

Positional Parameters – &REST

- ◊ **&rest** gives us a way of writing functions that handle any number of arguments

  (defun func (a b c &rest x) ...)

- ◊ For example the following calls can be used

  (func 1 2 3) – must have at least 3 parameters – nil -> x
  (func 1 2 3 4) – (4) -> x
  (func 1 2 3 4 5) – (4 5) -> x
  etc.

- ◊ **&rest** should follow all the other positional parameters
  - » **It "absorbs" the parameters left over after all the other positional parameters have be assigned a value from the parameter list.**

Keyword Parameters

- Optional parameters are useful but can sometimes be too restrictive.

- Because they are positional must have "prior" parameters to pass any specific optional parameter

- For example

  (defun func (&optional a b) ...)

- Must pass a value for **a** if you want to pass a value for **b**

  (func 1) – 1 -> a nil -> b
  (func nil 1) – nil -> a 1 -> b – need nil to get b to be 1

- Use **keyword parameters** to get around this restriction
  - » **A keyword parameter is a named parameter**
  - » **Associate argument by name not position**

© Gunnar Gotshalks

Keyword Parameters – 2

- ◊ For example

```
(defun func (&key  name1  name2) ... )
```

- ◊ Can have the following calling sequences

```
( func )
     gives   nil -> name1    nil -> name2
( func  :name1  11 )
     gives   11 -> name1    nil -> name2
( func  :name2  22 )
     gives  nil -> name1    22 -> name2
( func  :name1  11   :name2  22 )
     gives  11 -> name1    22 -> name2
( func  :name2  22   :name1  11 )
    gives  11 -> name1    22 -> name2
```

Keyword Parameters – 3

- Keyword parameters can also have default and “passed” variables similar to optional parameters

- For example

```
(defun func (&key  (name1  default)
                   (name2  default  passed) ... )
```

Ordering of Parameter Designators

- Function parameters are ordered as follows
  - **required**
  - **&optional**
  - **&rest**
  - **&keyword**
- Note that keyword parameters become a part of the structure of the **&rest** parameter

```
(defun func (& rest  x  &key  y ) ( list  x  y) )
(func  :y  7 )  ==>  ( ( :Y 7 )  7 )
```

- Keywords evaluate to themselves
  - **the value of :y is :y**

Closures - 1

- ◊ So far we have seen **local (bound)** and **global (free)** variables
- ◊ Variables can also be **dynamic** and **static**
- ◊ A **dynamic variable** is created on entry to a function and is disposed of on exit from a function
  - » **A standard effect for parameters in Lisp**
- ◊ A **static variable** is created once for a function and exists as long as the function definition exists
  - » **It retains its value between function calls**
  - » **It can keep track of information across function calls**

Closures - 2

- ◊ In Lisp static variables are associated with a function as a **lambda-closure**
  - » **Free symbols in a lambda expression are bound to the values of an enclosing function’s parameters**
  - » **And the lambda expression is returned as the value of the enclosing function**

Closures - 3

- ◊ An example – an even number generator
  - **Every call of the generator returns the next even number**

```
(defun egen (*aNumber*)
  (function
       (lambda() (setq *aNumber* (+ *aNumber* 2)))
))
```

- ◊ Create an instance of the generator

```
(setq gen1 (egen 0))
```

- ◊ Use the generator

```
(funcall gen1)  ==> 2
(funcall gen1)  ==> 4
(funcall gen1)  ==> 6
```

Closures - 4

- What does the definition of the closure look like?
- In gcl you can see something like the following

```
(LAMBDA-CLOSURE ((*ANUMBER* 0))   ; free variable
    () ((EG BLOCK #<@00218F50>))
    ()                                ; arguments lambda function
   (SETQ *ANUMBER*
           (+ *ANUMBER* 2))  ; body of the lambda function
)
```

- This is identical to how functionals such as bu are created
- When the lambda function in egen is evaluated its free symbol *ANUMBER* is bound to a location given by the closure. The value of *ANUMBER* was initialized to zero at the time of creating the closure

Multiple Function Closures

- ◊ You can have multiple functions within a single closure
- ◊ For example we want to be able to get the next even or the next odd number in a sequence
- ◊ This requires having two functions
  - » **One to get to the next even number**
  - » **The other to get to the next odd number**
  - » **Both functions must share the same “last value generated”**

Multiple Function Closures – 2

◊ Generate even-odd numbers in increasing sequence

```
(defun eog (*seed*)
 ( list (function (lambda()
    (setq *seed* (cond ((evenp *seed*) (+ *seed* 2))
                             ( t   (1+ *seed*))))))
         (function (lambda()
            (setq *seed* (cond ((oddp *seed*) (+ *seed* 2))
                                     ( t   (1+ *seed*))))))
    ))
```

◊ **eog** returns a list of two function definitions with a common global ***seed***

EOG use example

- ◊ **(setq eogFns (eog 0))** **; Create an instance of both functions**

  (setq fn1 (first eogFns)) **; Create the next even function**

  (setq fn2 (second eogFns)) **; Create the next odd function**

- ◊ Use **fn1** and **fn2** as in the following

```
(funcall fn1)  ==>  2
(funcall fn1)  ==>  4
(funcall fn2)  ==>  5
(funcall fn2)  ==>  7
(funcall fn1)  ==>  8
```

- ◊ To see the lambda-closures in Clisp use **(symbol-function ‘eogFns)**. Can also look at the values for **fn1** and **fn2.**