Writing More Flexible Functions

parameter designators and closure
(see Wilensky Chapter 12)

Shakil M. Khan
adapted from Gunnar Gotshalks

motivation

- so far, only saw how to define ordinary functions
- but earlier, we used functions that can take:
  - optional arguments
    (get 'purse 'change 'unknown)
  - variable number of arguments
    (+ 1 2), (+ 23 45 3 55 7 …)
- we can define such functions using parameter designators

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 languages – Java, C, Pascal, Turing, etc.
- e.g.

  (defun func (**x y z**) (....)

  (func **a b c**)

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

positional parameters – &OPTIONAL

- &optional provides a means of adding optional parameters
- e.g.: (defun func (a b &optional x y z) ...)
- 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 two optional parameters x and y:

  (defun func (a b &optional x (y yDefault)…)
- can set default values for optional parameters
  - if an argument is not given then the default value of x is nil
  - if an argument is not given then the default value of y is yDefault

&OPTIONAL details (cont'd)

- for example, consider:

  (defun func (a b &optional x (y 5)…)
- then calling the function with
  - (func 1 2)

    gives 1 → a, 2 → b, nil → x, 5 → y
  - (func 1 2 3)

    gives 1 → a, 2 → b, 3 → x, 5 → y
  - (func 1 2 3 4)

    gives 1 → a, 2 → b, 3 → x, 4 → y

&OPTIONAL details (cont'd)

- 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
- e.g. 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) ...)

- e.g. 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

example of &REST

- want to define sum that can take any number of arguments (using binary +)
- (defun sum (&rest l)

```
    (cond ((null l) 0)
          (t (+ (car l) (apply 'sum (cdr l))))
    )
)
```

- (sum 1 2 3) → 6, here l = (1 2 3)
- (sum 1 2 3 4 5) → 15

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
- e.g.:
  - (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

keyword parameters (cont'd)

- use keyword parameters to get around this restriction
  - a keyword parameter is a named parameter
  - associate argument by name not position
- e.g.:
  - to define
    (defun func (&key name1 name2) ...)
  - to call/invoke
    (func :name2 55)

keyword parameters (cont'd)

- (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 (cont'd)

- keyword parameters can also have default and passed variables similar to optional parameters
- e.g.:

  (defun func (&key name1 (name2 default2)
  (name3 default3 passed3) …)

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

- 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 - a motivational example

- an even number generator
  - every call of the generator returns the next even number
  - one solution – use a global variable
    (defun egen () (setq *seed* (+ *seed* 2)))
    (setq *seed* 0)
    (egen) → 2
    (egen) → 4

closures example (cont'd)

- but:
  - another function/we might accidentally clobber *seed* in between calls
  - can't use egen to generate 2 sequences of even numbers simultaneously
- another solution – use static variables

closures (cont'd)

- in Lisp static variables are associated with a function as a lambda-closure
  - create closure of a function by supplying the function \`function' with a lambda expression as argument
  - if the free symbols of that lambda expression are bound to the values of an enclosing function's parameters, \`function' will return a "snapshot" of the function denoted by the lambda expression
  -
```
(defun egen (*seed*)
    (function
     (lambda() (setq *seed* (+ *seed* 2)))
))
```


closures (cont'd)

- …\`function' will return a "snapshot" of the function denoted by the lambda expression…
  - contains its own copy of symbols that are free in the lambda expression (e.g. *seed*)
  - no code can access this variable except the code that comprises the closure
    - the variable is closed-off to the outside world (and hence the name \`closure')

closures example (cont'd)

- create some instances of the generator

  (setq gen1 (egen 0))
  (setq gen2 (egen 0))

- use the generator
  - (funcall gen1) → 2
  - (funcall gen1) → 4
  - (funcall gen1) → 6
  - (funcall gen2) → 2
  - (funcall gen2) → 4
  - (funcall gen1) → 8

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


multiple function closures

- you can have multiple functions within a single closure
- e.g.: 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
  - another to get to the next odd number
  - both functions must share the same "last value generated"

multiple function closures (cont'd)

- 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 (car eogFns)) ;create the next even function
- (setq fn2 (cadr 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

Macros

(see Wilensky Chapter 13 & 14.5)

Shakil M. Khan

adapted from Gunnar Gotshalks

motivation

- consider writing support functions to extract the components of a data structure describing a circle

  ((xPosition yPosition) radius)
- the following could be introduced

```
(defun xPos ( cir ) ( caar cir ))
(defun yPos ( cir ) ( cadar cir ))
(defun radius ( cir ) ( cadr cir ))
```

- advantages:
  - makes program more readable
  - insulates program from low level implementation decisions – useful since we only need to update these function definitions if our circle data structure changes

motivation (cont'd)

- but this is not too efficient
  - involves additional overhead of new function call
  - much more costly than our original caar / cadar / cadr
- we want a convenient way of expressing what we want to say, but avoid any additional overhead
  - we can do this with a macro

what are macros?

- macros are functions that output program text
  - they output a sequence of characters which Lisp handles as if you had typed them in yourself
  - the sequence is analyzed within the context in which it appears
  - eventually the sequence is part of an expression that will be evaluated (hence the term program text)
- in the simplest case a macro is like an abbreviation
  - **ASAP** ==> **A**s **S**oon **A**s **P**ossible

why use macros?

- can modify the syntax of writing expressions in Lisp (within the bounds of a list structure)
- you write an expression in a syntax that
  - is more expressive of the intent
  - frequently requires less typing
  - is easier to understand
  - is less error prone

example 1 : circle support functions (macros)

- convert the functions to macros.
  - (defmacro xPos (cir) (list 'caar cir))
    (xPos cir) returns (caar cir)
  - (defmacro yPos (cir) (list 'cadar cir))
    (yPos cir) returns (cadar cir)
  - (defmacro radius (cir) (list 'cadr cir))
    (radius cir) returns (cadr cir)
- note the use of list to combine terms and the use of quote to not evaluate constants

use the macros

- at the interpreter level

  (setq aCircle '((11 22) 33))
  (xPos aCircle)
  > 11
- no difference is visible between the macro and the
- the xPos macro was expanded and then evaluated because of the interpreter read-eval-print loop
- use the following to see the output of xPos

  (macroexpand '(xPos aCircle))
  → (caar aCircle); ← cir replaced with symbol aCircle
  → t
- the output of xPos is evaluated by eval

example 2

- illustration that macros output program text
- consider the following
  - (defmacro xx () '(+ a b))
    (defun xy (a b) (xx))
    (xy 3 4) → 7
  - (defun xx () (+ a b))
    (defun xy (a b) (xx))
    (xy 3 4) → "a is unbound"

macro expansion

- if you try, in Clisp, (symbol-function 'xy) you will see in each case that xx remains as a symbol; only at evaluation time is xx evaluated
  - a macro is expanded and then evaluated
  - macros are expanded once within a function; once expanded the program text remains within the body of the function.

construction method

- use list to build the lists and sublists, etc., corresponding to the s-expression you want to build
- use quote to insert constants – such as function names – leave variables unquoted so they evaluate
  - free symbols within the macro are "constants" as far as macro creation is concerned so they must quoted

    ```
    (defmacro xx () ( list '+ 'a 'b ) )
    ```

    +, a and b are free symbols so they are quoted;
    in this case we want the entire list, so alternately could have:

    ```
    (defmacro xx () '( + a b ) )
    ```

construction method (cont'd)

- bound symbols within the macro are usually not quoted because you want the symbols within the macro expression to bind to the local symbols with the same name
  - (defmacro xPos (cir) (list 'caar cir))
  - cir is a bound symbol so leave it unquoted – we want it to bind to the parameter cir
  - caar is a constant

example 3

- write a macro foreach that applies a function (func) to each item in a list that satisfies a predicate (test) and returns nil otherwise

```
( setq b1 '( 11 20 33 40 55 60 ) )
( foreach b1 '1+ 'evenp )
            → (nil 21 nil 41 nil 61)
```

- first write the expression we want identifying the parameters

```
( mapcar ( function ( lambda ( item )
                    ( cond ( ( funcall test item )
                             ( funcall func item ))))))
              list
)
```

example 3 (cont'd)

- create the macro definition header

```
(defmacro foreach ( list func test )
```

- write the body of the macro – for every '(' you need '(list':

```
(list 'mapcar
  (list 'function
    (list 'lambda
          (list 'item)
          (list 'cond
                (list (list 'funcall test 'item)
                      (list 'funcall func 'item)))))
   list )
) ;; end foreach
```


example 3 (cont'd)

- after you have loaded the macro definition you can see its expansion
  - (macroexpand-1 '(foreach b1 '1+ 'evenp))
- the result is the program text we expect
  - ```
    (mapcar ( function ( lambda ( item )
                       ( cond ( ( funcall 'evenp item )
                                ( funcall '1+ item )))))
            b1)
    ```
- if we enter at the interpreter level the macro expands and evaluates
  - (foreach b1 '1+ 'evenp)

    → (nil 21 nil 41 nil 61)
    ```

parameters for macros

- macro parameters can be designated as &optional, &keyword and &rest (in macros a synonym is &body)
- &optional and &keyword can have default values and the passed flag (see the slides on *Writing More Flexible Functions*)
- the example shown uses list but you can invoke any functions you want to compute the final program text the macro creates

flexible parameter structures

- so far we haven't seen any flexibility in the syntax of parameter structures
- suppose we want to call the macro using the following
  - (foreach b1 (apply '1+) 'evenp)
  - note: within a macro the arguments are not evaluated; the text itself is passed!

flexible parameter structures (cont'd)

- the corresponding macro definition could be the same as before except we extract the function '1+ from the list (apply '1) passed to func

```
(defmacro foreach ( list func test )
   (list 'mapcar
     (list 'function
       (list 'lambda
              (list 'item)
              (list 'cond
                     (list (list 'funcall test 'item)
                       (list 'funcall ( cadr func ) 'item)))))
     list )
)
```


flexible parameter structures (cont'd)

- suppose we want the following prototype calling sequence
  - (foreach (item in list) (apply func) (when test))
- you can see how the macro definition would get more complex
- there is additional syntax that can be used in macro definitions to simplify writing macros

flexible parameter structures (cont'd)

- the flexibility can be achieved with the following

```
(defmacro foreach ( ( item in list )
                    ( apply func )
                    ( when test ) )
   (list 'mapcar
     (list 'function
       (list 'lambda
             (list 'item)
             (list 'cond
                   (list (list 'funcall test 'item)
                         (list 'funcall func 'item)))))
     list )
 )
```

- same body as original! Parameter list has a structure

flexible parameter structures (cont'd)

- macros for foreach could have any of the following calling sequences and still retain the same body
  - **(foreach list func test)**
  - **(foreach (item in list) (apply func) (when test)**
  - **(foreach (list (when test) do func))**
  - **(foreach list (when test) func)**
  - **(foreach item in list do func when test)**
- the "real parameters" are the **green names** in the above
- use names at any level to extract corresponding arguments when the macro is called
- the other names in the parameter list are "**noise words**" inserted to make the macro prototype easier to understand and the macro call easier to read.

noise words

- noise words are exactly that – noise; they have no meaning; you can have anything in their place
- e.g. assume you define a macro with the following header:

```
(defmacro foreach
      ((item in list) (apply func) (when test)) .... )
```

- then you can call it with any of the following and have the same semantics:

```
(foreach (item in b1) (apply '1+) (when 'evenp))
(foreach (xyz abc b1) (gortz '1+) (NOTWHEN 'evenp))
(foreach ((1 2 3) in b1) (doNOTapply '1+) (NEVER 'evenp))
```

- but you have to know which "names" are noise

simpler macro definitions

- using list is a complex way of constructing macro
- the same macro can be defined with the following

```
(defmacro foreach
      ((item in list) (apply func) (when test))
 `(mapcar
 #'(lambda (item)
      (cond ((funcall ,test item)
                  (funcall ,func item))))
      ,list)
 )
```

note use of backquote and comma

simpler macro definitions (cont'd)

- notice this style is very similar to the function definition

```
(defmacro foreach
     ((item in list) (apply func) (when test))
`(mapcar
     #'(lambda (item)
              (cond (( ,test item) (funcall ,func item))))
     ,list)
)
```

- ` (backquote) is similar to quote – the quoted expression is returned
- use , (comma) to "unquote" parameters – use the value of the parameter

use of comma

- "," evaluates the following form:
  - (defmacro comma-1 (aPair)
    `(cons ,(car aPair) ,(cdr aPair))
)
  - macroexpand-1 on (comma (a b)) gives the following
    (cons a (b))
- ",@" evaluates the following form and removes the outer ():
  - (defmacro comma-1 (aPair)
    `(cons ,(car aPair) ,@(cdr aPair))
)
  - macroexpand-1 on (comma (a b)) gives the following
    (cons a b)

a complex example

- first consider a useful function:

```
(let ((var1 val1) (var2 val2) … (varN valN))
      s-exp1
      s-exp2
      …
      s-expM
)
```

- creates local variables var1…varN, then evaluates s-exp1…s-expM (which contains references to var1…varN)

a complex example (cont’d)

- a contrived example to show various features working together:

```
(defmacro complex (&body items)
   (let ((func (bu ‘cons 0)))
     `(list ,@(mapcar #’( lambda ( anItem )
                             `(funcall ,func `,anItem))
               items))
))
```

- macroexpand-1 on (complex A B C) gives the following

```
(list (funcall #<function> ‘A)
(funcall #<function> ‘B)
(funcall #<function> ‘C))
```

- when executed the following results

```
( ( 0 . A ) ( 0 . B ) ( 0 . C) )
```

on writing macro definitions

- when writing macro definitions you can combine techniques
  - use parameter names within structures to extract components
  - use full Lisp functionality to compute program text
  - use backquote to simplify the construction of program text structures that do not require computation
    - use comma to unquote parameters

recursive macro definitions

- write a macro that will cons all the items in a list
  (sc a b c d) →
  (cons a (cons b (cons c (cons d nil))))
- here is a naive attempt

```
(defmacro sc ( &body theList )
   (cond ( ( null theList ) nil )
             ( t ( list 'cons ( car theList )
                                   ( sc ( cdr theList ))))
))
```

- but you get stack overflow on expansion; why?

recursive macro definitions (cont'd)

- putting (print theList) after the t you get
  - (a b c d)
  - ((cdr thelist))
  - ((cdr thelist))
  - … forever – a very long time
- recall the arguments given a macro are text
  - arguments are not evaluated
  - so (cdr theList) is passed not the result of cdr

recursive macro definitions (cont'd)

- recursive macro definitions require helper functions

```
(defmacro sc ( &body theList )
    ( recursiveHelper theList ) )

(defun recursiveHelper ( theList )
    ( cond ( ( null theList ) nil )
           ( t ( list 'cons ( car theList )
                   ( recursiveHelper ( cdr theList ))))))
```

- now

```
( sc a b c d ) →
    ( cons a ( cons b ( cons c ( cons d nil ))))
```

- most macros are not so complex as to require recursion, but the technique is available when necessary

info

- reminder: assignment/report 1 due next week
  - hand in hard copy + submit soft copy (submit programs only, so that I can execute your code: submit 3401 r1 <files>)
- class test 1 on June 11 (approx. from 7:00 PM to 8:30 PM)