Macros

WIlensky Chapters 13 & 14

© Gunnar Gotshalks

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 analysed within the context in which it appears**
  - » **Eventually the sequence is part of an expression that will be evaluated**
    - > **Hence the term program text is used to describe the output of a macro**
- In the simplest case a macro is like an abbreviation
  - » **ASAP ==> As Soon As Possible**
  - » **We write ASAP but expect our readers to see this as being "As Soon As Possible"**

© Gunnar Gotshalks

Example – 1

- Consider writing support functions to extract the components of a data structure describing a disc in a plane

  > ((xPosition yPosition) radius)

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

- They could be introduced to make a program more readable by hiding the structure of a disc
- We can do the same thing with a macro

© Gunnar Gotshalks

Disc support functions ==> macros

- Convert the functions to macros.

```
(defmacro  xPos (disc )  ( list  'caar disc ) )
  > Returns   ( caar disc )
(defmacro  yPos (disc )  ( list  'cadar disc ) )
  > Returns   ( cadar disc )
(defmacro  radius (disc )  ( list  'cadr disc ) )
  > Returns   ( cadr disc )
```

- Note the use of list to combine terms and the use of quote to not evaluate constants

© Gunnar Gotshalks

Use the macros

◊ At the interpreter level

```
( setq  theDisc  '( (11 22 )  33 ) )
( xPos  'theDisc )
  > 11
```

◊ No difference is visible between the macro and the function – why?

◊ 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-1  '( xPos  theDisc ) )
  > ( caar  theDisc )
```

disc replaced with symbol theDisc

◊ The output of **xPos** is evaluated by **eval**

© Gunnar Gotshalks

Why use macros?

- Since the result of executing function version and macro version have same result, why have 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**
  - » **The corresponding macro produces an equivalent Lisp expression that is executed**

© Gunnar Gotshalks

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"
```

© Gunnar Gotshalks

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.**

© Gunnar Gotshalks

Macro Expansion – 2

◊ In **PowerLisp** and **Clisp** the function is modified even at the interpreter level.

```
(defmacro  xx  ()  '( +  a  b ) )
(defun  xy  ( a  b )   ( xx ) )
( xy  3  4 )  ==> 7
( defun  xx  ()   ( +  a  b ) )
( xy  3  4 )  ==>  7   (in PowerLisp, Clisp)
                   "a is unbound" (in gcl)
```

> **In some Lisps, such as gcl, at the interpreter level, the latest definition is always accessed**

◊ The macro is expanded only once if the function is invoked within a loop.

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 ) )
```

© Gunnar Gotshalks

Construction method – 2

> 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 ( disc )  ( list  'caar  disc ) )
```

> disc is a bound symbol so leave it unquoted – we want it to bind to the parameter disc. caar is a constant

© Gunnar Gotshalks

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
)
```

© Gunnar Gotshalks

Example 3 – 2

◊ 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
```

© Gunnar Gotshalks

Example 3 – 3

- 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 )
```

© Gunnar Gotshalks

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 slide set *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

© Gunnar Gotshalks

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!**

© Gunnar Gotshalks

Flexible parameter structures – 2

- 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 )
)
```

© Gunnar Gotshalks

Flexible parameter structures – 3

- 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.

© Gunnar Gotshalks

Flexible parameter structures – 4

◊ 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.

© Gunnar Gotshalks

Flexible parameter structures – 5

- 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.

© Gunnar Gotshalks

Noise words

◊ Noise words are exactly that – noise. They have no meaning. You can have anything in their place.

◊ For example 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

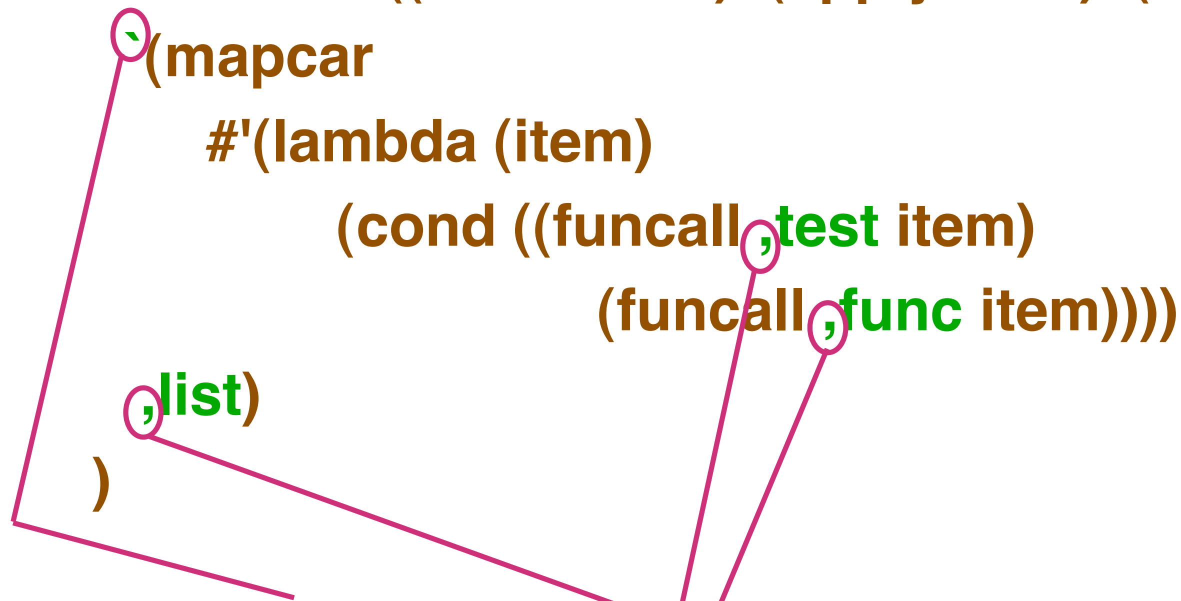
```
(foreach (item in b1) (apply '+) (when 'evenp))
(foreach (xyz abc b1) (gortz '+) (NOTWHEN 'evenp))
(foreach ((1 2 3) in b1) (doNOTapply '+) (NEVER 'evenp))
```

◊ But you have to know which "names" are noise.

© Gunnar Gotshalks

Simpler macro definitions – 1

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



© Gunnar Gotshalks

Simpler macro definitions – 2

◊ 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-1 (a b)) gives the following

```
( cons a (b) )
```

- ",@" evaluates the following form and removes the outer ().

```
(defmacro comma-2 ( aPair )
   `(cons ,(car aPair) ,@(cdr aPair))
)
```

  - macroexpand-1 on (comma-2 (a b)) gives the following

```
( cons a b )
```

© Gunnar Gotshalks

A Complex Example

◊ 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: lambda(y) (funcall f x y)> 'A)
    (funcall #<function: lambda(y) (funcall f x y)> 'B)
    (funcall #<function: lambda(y) (funcall f x y)> 'C)
))
```

» When executed the following results

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

© Gunnar Gotshalks

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**

© Gunnar Gotshalks

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?

© Gunnar Gotshalks

Recursive macro definitions – 2

◊ 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**

© Gunnar Gotshalks

Recursive macro definitions – 2

- 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

© Gunnar Gotshalks