Multiple Value Functions
(see Wilensky Chapter 16.4)

Shakil M. Khan
adapted from Gunnar Gotshalks

multiple value functions

- in other languages, one can return multiple values on one call
  - e.g. in C, using *call by reference* (pointers)
- but in Lisp, all parameters are passed by value
  - they cannot be changed
- to return multiple values in Lisp
  - need to construct and return a list of the results you want the function to return (this is costly!)
  - the caller must extract, through car and cdr the values of interest (inconvenient)
  - this occurs frequently enough that Lisp permits multiple values to be returned by some special functions/macros

catching multiple values

- by default, if a function returns multiple values
  - only one is passed back, the rest are discarded
  - unless you specifically ask for the other values
- e.g. (round aNumber) returns two values
  - the rounded value and
  - the value needed to add to the rounded result to get the original number
- (round aNumber) → roundedValue; difference
  where, difference = aNumber - roundedValue
- (round 7.6) → 8; -0.4
  - not a list! (car (round 7.6)) fails
  - (print (round 7.6)) → 8
    uses first value by default

catching multiple values (cont'd)

- use the following macro to create a list of multiple value returns
  - (multiple-value-list (round aNumber))
    → (roundedValue restoreNumber)
  - (multiple-value-list (round 7.6)) → (8 -0.4)
- can assign the values to symbols using the following macro
  - (multiple-value-setq (val diff) (round 7.6))
  - 8 → val and -0.4 → diff
  - note setq implies global symbol

catching multiple values (cont'd)

- can create a local context for variables instead of using global variables

```
(let ((val nil) (diff nil))
    (multiple-value-setq (val diff) (round 7.6))
    ;; ... use val and diff in list of forms
    (print val)
    (print diff)
    (print (+ val diff))
)
```


catching multiple values (cont'd)

- the following shows that let is syntactic sugar for a lambda function
  - ((lambda (val diff)

```
( (lambda ( val diff )
          (multiple-value-setq (val diff) (round 7.6))
          ;; ... use val and diff in list of forms
          (print val)
          (print diff)
          (print (+ val diff))
  )
  nil nil ;; initial values for val & diff
)
```

catching multiple values (cont'd)

- instead of using let which needs initial values for its parameters, can use the following

```
( multiple-value-bind (val diff ) (round 7.6 )
    ;; ... list of forms using val and diff ...
    ( print val )
    ( print diff )
    ( print ( + val diff ) )
)
```


catching multiple values (cont'd)

- consider the following:

```
(defun functionName (val diff)
        (print val) (print diff) (print (+ val diff))
)
```

- suppose we want to pass the values returned by round to functionName

```
(functionName (round 7.6 ))
```

- but ordinary cLisp argument handling mechanism only passes a single value from a form to a function

catching multiple values (cont'd)

- can use the following to pass the return values to a function
  - its arity equals the number of returned values
  - (multiple-value-call

```
              #'functionName (round 7.6 ))

(defun functionName (val diff)
              (print val) (print diff) (print (+ val diff))
)
```


generating multiple values

- the last form in a function is a call to values
  (values 1 2 3) → 1; 2; 3
- here is a function to tear a list into its first and rest parts

```
(defun unCons ( theList )
              (values ( car theList ) ( cdr theList ) )
)
(uncons '( a b c ) ) → a; ( b c )
```

- what about unconsing an entire list? use apply to strip the outer level of parenthesis
  (apply 'values '(a b c d e)) ==> a; b; c; d; e

generating multiple values (cont'd)

- can use values (with no arguments) to build a Lisp function that returns no values at all
  - the last statement of the function: (values)
  - useful if a function is used only for its side-effects
  - Lisp supplies a gratuitous value of nil if such a function is used in a context where it is not anticipated
    - e.g. (print (values)) → nil

consequences of passing around multiple values

- has some consequences on the design of a Lisp system
- recall: code that does not anticipate multiple values ignores all but the first value
- however, most built-in functions/macros that tends to return the values of other forms are set up to handle multiple values
  - e.g. cond, let, … passes multiple values
  - and, or passes multiple values from the last sub-forms
  - (and nil (values 3 4)) → nil
    (and t (values 3 4)) → 3; 4, (and (values 1 2) t) → t

Pattern Matching
(see Wilensky Chapter 21)

Shakil M. Khan
adapted from Gunnar Gotshalks

pattern matching

- a ubiquitous function in intelligence is pattern matching
  - e.g. IQ tests often contain pattern matching problems
- pattern matching means to compare one object with another object and recognize if they are similar
  - basic case is comparing constants
  - more interesting is to compare parameterized patterns
    - > A is like B except for ....
    - > A is like B where ...

  (a statement that sub-objects, while not identical, correspond to each other)

applications

- to classify data
  - whether it has a certain property
- AI applications
  - e.g. to extract information from natural language like text
  - "John hit Mary" → (hurt Mary)

criterion for similarity

- when can we consider two patterns to be similar?
  - exact match
  - the first two items are the same
  - the last item is the same
  - …
- we want to build a generic pattern matcher
  - can't decide in advance what criterion to use
  - rather, provide a fairly general scheme and later implement different criteria for different applications
  - one way to achieve this → introduce pattern matching variables

what is a pattern?

- in Lisp, a pattern is a form (s-expression) that contains
  - constants – called literals
  - pattern matching variables
- we need a syntax to differentiate the two
  - can prefix pattern matching variables with ?
  - e.g. ?x ?abc
- an abstract pattern could look like
  - (a b ?x c ?y)
- a more meaningful pattern could be
  - (causes (hit ?x ?y) (hurt ?y))
  - interpreted as – x hitting y, causes y to be hurt

pattern variable representation

- how will we represent pattern matching variables in Lisp?
  - the rest is simply a list with symbols for the constants
- use the construct (*VAR* X)
  - where *VAR* is a special symbol we recognize within the matcher program

when do two patterns match?

- two patterns can be matched when it is possible to unify them
- unification (a term borrowed from theorem proving) means an assignment can be made to the variables in each pattern such that the patterns become identical
  - we usually mean the most general possible assignment
- an assignment is shown by the pair
  (variable value)
  - ((*VAR* X) abc)
  - ((*VAR* X) (*VAR* Y))

unification examples – 1

- (a ?x b)
  (a y b)
  match if ?x <-- y
  we say that ?x is bound to y
- (a ?x b)
  (a ?y b)
  match if ?x <-- ?y
- (a ?x (b ?z))
  (a (((e))) ?y)
  match if ?x <-- (((e)))
  ?y <-- (b ?z)

unification examples – 2

- more complex examples
  - (a ?x ?x) match if ?x = ?y
    (a ?y c) and ?y = c
    - cannot naively bind ?x to ?y and then ?x to c as then we are trying to assign two different values to ?x
    - need to substitute ?y for ?x and then see that ?y binds to c
  - (a ?x ?x ?x)
    (a ?y ?y ?y)
    - cannot naively try to bind ?x to ?y, as on the second attempt, we end up binding ?y to ?y, then on the third attempt, we have an infinite loop

unification examples – 3

- more complex examples
  - (a ?x ?x) there is no consistent
    (a ?y (b ?y)) binding to make a match
    - again need to prevent an infinite loop

pattern variable input

- how do we represent input?
  - we would like to keep the notation ?x
  - instruct the read program to recognize the construct ?symbol and create the list (*VAR* symbol)

    (set-macro-character #\? ;;see page 245
      #'(lambda (stream char)
            (list '*var* (read stream t nil t))))
  - test with (read), enter '?x and see (*VAR* x) as the result

pattern matcher output

- need to distinguish three cases (see p369 for a discussion)
  - no match is possible
    → output is nil
  - match is possible but no variable bindings are required
    → output is T ; nil – two values returned
  - match is possible with variable bindings
    → output is T ; (list of bindings)
  - a binding is a pair ((*VAR* variable) value)
- example with a binding required
  - (match '(a ?x c ?y e) '(a b ?z d e))
  → T ; (((*VAR* Y) D) ((*VAR* Z) C) ((*VAR* X) B))

matcher

- reminder that we need to define the macro characer ?

```
(set-macro-character #\?
  #'( lambda ( stream char )
        ( list '*var* ( read stream t nil t) )))
```

- the entry function creates the initial empty binding

```
(defun match ( pattern1 pattern2 )
  (match-with-bindings pattern1 pattern2 nil ))
```


matching cases

- matching two patterns requires a recursive descent into the patterns to match sub-patterns; the following cases can occur
  - pattern1 – a variable, an atom, a list
  - pattern2 – a variable, an atom, a list

matching cases (cont'd)

- the matching program has to examine the possible combinations
- pattern1 pattern2 result

pattern1	pattern2	result
atom	atom	match if equal, else no match
atom	variable	try to bind atom to variable
atom	list	no match
variable	atom	try to bind atom to variable
variable	variable	try to bind variable to variable
variable	list	try to bind list to variable
list	atom	no match
list	variable	try to bind list to variable
list	list	recursive descent on first and rest

match with bindings

- organize when bindings need to be done
- (defun match-with-bindings (pattern1 pattern2 bindings)

```
    (cond
        ;; pattern 1 is a variable?
            ( ( pattern-var-p pattern1 )
              ( variable-match pattern1 pattern2 bindings) )
        ;; pattern 2 is a variable?
            ( ( pattern-var-p pattern2 )
              ( variable-match pattern2 pattern1 bindings ) )
        ;; pattern 1 is an atom? note use of values
            ( ( atom pattern1 )
              ( if ( eq pattern1 pattern2 ) ( values t bindings)))
        ;; pattern 2 is an atom?
            ( ( atom pattern2 ) nil )
```

match with bindings (cont'd)

```
;; pattern1 and pattern2 are both lists – use recursion and
   multiple values
    ( t
      (multiple-value-bind ( flag carbindings )
            (match-with-bindings   ( car pattern1 )
                                   ( car pattern2 )
                                   bindings )
            (and flag
                 (match-with-bindings      ( cdr pattern1 )
                                           ( cdr pattern2 )
                                           carbindings )
    )))))
```


variable match

- find a binding for pattern-var within item using the current bindings
- (defun variable-match (pattern-var item bindings)

```
;; check for equality – no additional bindings are
   necessary
   (if (equal pattern-var item) (values t bindings)
;; otherwise ...
```

variable match (cont'd)

- need a binding

```
(let ((var-binding ;; determine if a binding already exits
      (get-binding pattern-var bindings)))
;; handle the case where a binding exists
      (cond    (var-binding
                 (match-with-bindings var-binding item bindings))
;; no binding for the variable – check for circularity –
   need to see if the pattern-var occurs in item or is
   bound to a variable in item
               ((not (contained-in pattern-var item bindings))
                (values t
                        (add-binding pattern-var item bindings)))
      )
)))
```


contained-in

- check for circularity by – seeing if pattern-var occurs in item or is defined as the value of a binding of a variable in item

```
(defun contained-in (pattern-var item bindings)
   ;; cannot be contained in an atom
      (cond ((atom item) nil)
   ;; check if item is a variable
      ((pattern-var-p item)
   ;; does pattern-var occur in item
      (or      (equal pattern-var item)
   ;; does pattern-var occur as the value of a binding?
               (contained-in pattern-var
                             (get-binding item bindings)
                             bindings)))
```

contained-in (cont'd)

```
;; the item is a list so recursively check for contained-in
  (t
   (or     (contained-in pattern-var (car item)
                          bindings)
           (contained-in pattern-var (cdr item)
                          bindings)
  ))
))
```


matcher – housekeeping functions

- add the binding to the current bindings (a list of 2 element lists)

```
(defun add-binding ( pattern-var item bindings )
  ( cons ( list pattern-var item ) bindings ))
```

- if item is a pattern variable return true, else return false

```
(defun pattern-var-p ( item )
  ( and ( listp item ) ( eq '*var* ( car item ))))
```

- get the binding, if any, for pattern-var in the binding list bindings

```
(defun get-binding ( pattern-var bindings )
  ( cadr ( assoc pattern-var bindings :test #'equal)))
```

info

- class test 1 on June 11
  - closed book exam
  - syllabus: everything covered up to and including multiple value functions
  - bring York photo ID
- good luck!