TUTORIAL 2

CSCI3230 (2013-2014 First Term)

By Paco WONG (pkwong@cse.cuhk.edu.hk)

Outline

- Predicates
- Conditional constructs
- Iteration
- Function
- Macro
- Structure
- Property list

Type Predicates

General

Function	**Meaning**
TYPEP *object data_type*	Is the type of *object data_type*?
SUBTYPEP *subtype data_type*	Is the subtype a *subtype* of *data_type*?

Specific

Function	**Meaning (Is the argument data_type?)**
NULL *argument*	Is the argument ()?
ATOM *argument*	Is the argument an atom (not a cons)?
SYMBOLP *argument*	Is the argument a symbol?
CONSP *argument*	Is the argument a cons?
LISTP *argument*	Is the argument a list?

Example 1

```
> (symbolp 10)
NIL
> (atom 10)
T
> (typep 10 'atom)
T    ;same as the last result
```

Equality Predicates

Function	Meaning
EQ *x y*	Are the two objects implementationally identical? Same memory location
EQL *x y*	Are the two objects conceptually identical? EQ x y <u>OR</u> Number of same type with the same value <u>OR</u> Character objects of the same character
EQUAL *x y*	Are the two objects isomorphic? Same printed representations
= *x y*	Numbers only and ignore type

<u>Example 2</u>

```
> (setq q1 '(abc 123))
(ABC 123)
> (setq q2 '(abc 123))
(ABC 123)
> (eq q1 q2)
NIL
```

```
;cont'
> (eql q1 q2)
NIL
> (eql (car q1) (car q2))
T
> (equal q1 q2)
T
```

Conditional Constructs

- **AND {*form*}***
- If either one of the forms is NIL, returns NIL;
- Otherwise, returns the value of the last form.
- **NOT *argument***
- If the argument is NIL, returns T;
- Otherwise, returns NIL.
- **OR {*form*}***
- Evaluating the forms in order, if one of the forms is non-NIL, immediately returns that value;
- Otherwise, returns NIL.

Conditional Constructs

- ***IF test then [else]***
- Evaluates a ***test*** form
- If the result is non-NIL, next form is ***then***.
- If the result is NIL, next form is the optional ***else***.

Example 3

```
> (setq p 10)
10
> (setq d 3)
3
> (if (eql (mod p d) 0)
        (list '* p d)
        (list '+ (list '* d (floor (/ p d))) (mod p d)))
(+ (* 3 3) 1)
```

To enclose multiple forms in a form, use **(PROGN {*form*}*)**

Conditional Constructs

- **<u>COND { (*test* {*form*}*) }*</u>**
- Evaluates the ***test*** form in order
- If the value of *test* form is non-NIL
  - the next set of ***form***s is evaluated and returns its result.
- If none of the *test* forms non-NIL, returns NIL.

<u>Example 4.1</u>

```
> (setq person 'Sam)
SAM
> (cond
        ((eql person 'Peter) 'Boss)
        ((eql person 'Sam) 'Staff1001)
        ((eql person 'Paco) 'Staff1002)
        (t 'Intruder!))     ;Otherwise
STAFF1001
```

<u>Example 4.2</u>

```
> (setq person 'John)
JOHN
> (cond
        ((eql person 'Peter) 'Boss)
        ((eql person 'Sam) 'Staff1001)
        ((eql person 'Paco) 'Staff1002)
        (t 'Intruder!)       ;must be eval
        (t 'Alarm))          ;ignore
INTRUDER!
```

Indefinite Iteration

- **LOOP {*forms*}***
  - Repeatedly evaluates a series of forms

Example 5

```
>(setq x 3)
>(loop
    (print x)
    (setq x (- x 1))
    (if (zerop x) (return "Go!")))
3
2
1
"Go!"
```

Specific Iteration: DO

- **DO ({ (*variable* [*initial-value* [*step-form*]]) }*)**
  (*end-test* {*result*}*)
  {*declaration*}*
  {*tag* | *statement*}*)
- Repeatedly evaluates a series of forms until the end-test returns non-NIL.

Specific Iteration: DO

Example 6

```
>(setq x 3)
>(do ((a x (- a 1)))
    ((zerop a) "Go!")
    (print a))
3
2
1
"Go!"
```

variable ***initial-value*** ***step-form***

(DO ((a x (- a 1)))

end-test ***result***

((zerop a) "Go!")

statement

(print a))

Specific Iteration: DO*

- **<u>DO* ({ (*variable* [*initial-value* [*step-form*]]) }*)</u>**
  <u>(*end-test* {*result*}*) {*declaration*}* {*tag* | *statement*}*)</u>
- Looks exactly the same except that the name DO is replaced by DO*
- Except that the bindings and steppings of the variables are performed sequentially rather than in parallel

Try the following expression.

```
(dolist (x '(c u h k) x) (print x))
```

Pay attention to the last line of the output.

Let

- **<u>LET ({*var* | (*var value*)}*) {*declaration*}* {*form*}*</u>**
- Execute a series of forms **{*form*}*** with specified variables **{*var* | (*var value*)}***

<u>Example 7</u>

```
> (let ((x 3))                                ;don't forget the ()
          (loop (print x)
            (setq x (- x 1))
            (if (zerop x) (return "Go!"))))

3
2
1
"Go!"
```

YOUR CONTROL CONSTRUCTS

Much more powerful than #define macros in C

Named Function

- **DEFUN *name lambda-list {declaration|doc-string}* {form}***
- The *name* is a global name
- *lambda-list* specifies names for the *parameters* of the function

Example 8

```
> (defun decompose(p d)
        (if (eql (mod p d) 0)
          (list '* p d)
          (list '+ (list '* d (floor (/ p d))) (mod p d))))
DECOMPOSE
> (decompose 10 4)      ;the decompose function can be called globally
(+ (* 4 2) 2)
```

Lambda Expression

- **<u>LAMBDA *lambda-list . body*</u>**
- Defines an unnamed function
- Useful when you are going to use a function only once

<u>Example 9</u>

```
> ((lambda (p d)
              (if (eql (mod p d) 0)
                (list '* p d)
                (list '+ (list '* d (floor (/ p d))) (mod p d)))) 10 4)
(+ (* 4 2) 2)
```

<u>Example 8</u>

```
> (defun ~~decompose~~(p d)
              (if (eql (mod p d) 0)
                (list '* p d)
                (list '+ (list '* d (floor (/ p d))) (mod p d))))
DECOMPOSE
> (decompose 10 4)
(+ (* 4 2) 2)
```

Block and Exits

- <u>**BLOCK *name {form}****</u>
- Provide a structured lexical non-local exit facility.
- Example: Exit a loop until it hits a return (see Example 5)

<u>**Example 10**</u>

```
>(block blk-A (block blk-B (return-form blk-A 10) 20)) ;cascaded
10                                                     ;not 20
>(block nil (return 27)) ;the name is nil, same as (return-from nil (return 27))
27
```

Block and Exits

- The body of a defun form implicitly enclosed in a block.
- The block has the same name as the function.

Example 11

```
>(defun find-food (trapp)
               (format t "see the cheese. ")
               (if trapp
                   (return-from find-food (format t "there is a trap. turn away.")))
                (format t "eat it greedily."))
>(FIND-FOOD t)
see the cheese. there is a trap. turn away.
NIL
```

Macro

- **DEFMACRO name lambda-list [[{declaration}* | doc-string]] {form}***
- Its parameters are not evaluated before passing to the marco. They are evaluated in place of the macro call.

Example 12

```
> (defmacro macro_power (f) `(let ((x 1) (y 2)) ,f))    ; Back quote ` and comma ,
MACRO_POWER
> (macro_power (+ y 2))
4
> (macro_power (+ x y))
3
> (macroexpand-1 '(macro_power (+ y x)))            ;Expansion of the above
(LET ((X 1) (Y 2)) (+ Y X))                         ;Substitution takes place
T
```

YOUR DATA TYPE

Structure

- **DEFSTRUCT {structure-name | (slot-name {options}*)} [doc-string] {slot-description}+**
- Defines a new data type whose instances have named slots
- Arranges for setf to work properly on such access functions
- Functions associated with structure of name *sname*
  - Access function for each slot *slotname*: *sname-slotname*
  - Predicate: *sname*-p
  - Constructor function: make-*sname*
  - Copier function: copy-*sname*
- Other options
  - :conc-name, :constructor, :copier, :predicate, :include, :print-function, :type, :named, :initial-offset

Structure

Example 13

```
> (defstruct classroom (chair 30) (whiteboard 2) (computer 1))
CLASSROOM                                        ;new structure

> (setq shb504 (make-classroom))                 ;constructor
#S(CLASSROOM :CHAIR 30 :WHITEBOARD 2 :COMPUTER 1)

> (setq erb1009 (make-classroom :chair 50))      ;assign new value to slot
#S(CLASSROOM :CHAIR 50 :WHITEBOARD 2 :COMPUTER 1)

> (classroom-computer erb1009)                   ;access function
1

> (setf (classroom-whiteboard erb1009) 3)        ;assign new value with setf
3

> erb1009                                        ;see the result
#S(CLASSROOM :CHAIR 50 :WHITEBOARD 3 :COMPUTER 1)
```

Property List

- A kind of tabular structure
- Entries are associated with an indicator
- An object with unique identity (destructive operations)
- <u>**GET *symbol indicator &optional default***</u>
- Searches the property list of *symbol* for an indicator eq to *indicator*. If one is found, then the corresponding value is returned; otherwise default is returned.

<u>**Example 14**</u>

```
> (setf (get 'Tony 'Mobile) 'XIAOMI)
XIAOMI
> (get 'Tony 'Mobile)
XIAOMI
> (setf (get 'Tony 'CallNo) '46784655)
46784655
> (symbol-plist 'Tony)                    ;Show the whole property list
(CALLNO 46784655 MOBILE XIAOMI)
```

Summary

- S-expression
  - Atom and list
- Evaluation and its control
  - Form, QUOTE and EVAL
- Binding variable explicitly
  - SET, SETQ and SETF
- Cons cell and list
  - CONS, CAR, CDR, NTHCDR, NTH, APPEND, …
- Lispbox
- Predicates
  - TYPEP, SUBTYPEP, EQL, …
- Conditional constructs
  - IF THEN ELSE, COND, …
- Iteration
  - LOOP, DO, …
- Let
- Function
  - DEFUN
- Macro
  - DEFMACRO
- Structure
  - DEFSTRUCT
- Property list
  - GET

Feedback: email or http://goo.gl/5VLYA

Try it yourself

1. Write a recursive function *Fibonacci(n)* that returns the n^{th} number in the Fibonacci sequence. One reclusive version and iterative version.
2. Given a list of length n, we want to remove the cons cells from j to j+1, where j is from 0 to n-1.
3. Write a merge sort function for a list of number.
4. Use *mapcar* with unnamed and named function.

Hands on Lab
@SHB924
[W8 W9 H7 H8]

Hints

Exercise 1

http://www.cs.sfu.ca/CourseCentral/310/pwfong/Lisp/1/tutorial1.html

Exercise 2

```
> (setq l '(1 2 3 4 5 6 7 8 9 10 11 12))
(1 2 3 4 5 6 7 8 9 10 11 12)
> (nthcdr 3 l)
(4 5 6 7 8 9 10 11 12)
> (nthcdr 3 l)
(4 5 6 7 8 9 10 11 12)
> (setf (cdr (nthcdr 3 l)) (nthcdr 6 l))   ;because clisp doesn't allow setf on nthcdr
(7 8 9 10 11 12)
> l
(1 2 3 4 7 8 9 10 11 12)
```

Exercise 3

http://en.literateprograms.org/Merge_sort_%28Lisp%29

Suggested Readings

- Common Lisp the Language, 2nd Edition
  - http://www.cs.cmu.edu/Groups/AI/html/cltl/cltl2.html
- OnLisp
  - http://lib.store.yahoo.net/lib/paulgraham/onlisp.pdf
- Common LISP Hints
  - http://www.carfield.com.hk/document/languages/common-lisp-tutorial.html

Reference

- Common Lisp the Language, 2nd Edition
  - http://www.cs.cmu.edu/Groups/AI/html/cltl/cltl2.html
- http://lib.store.yahoo.net/lib/paulgraham/onlisp.pdf
- Common LISP Hints
  - http://www.carfield.com.hk/document/languages/common-lisp-tutorial.html
  - http://www.n-a-n-o.com/lisp/cmucl-tutorials/LISP-tutorial-19.html

- George F. Luger. Artificial Intelligence 5th ed. Addison Wesley.
- Wendy L. Milner. Common LISP, a Tutorial. Prentice Hall.

Appendix: Lambda-list

"

- A lambda-list has five parts, any or all of which may be empty:
- Specifiers for the ***required*** parameters. These are all the parameter specifiers up to the first lambda-list keyword; if there is no such lambda-list keyword, then all the specifiers are for required parameters.
- Specifiers for *optional* parameters. If the lambda-list keyword &optional is present, the ***optional*** parameter specifiers are those following the lambda-list keyword &optional up to the next lambda-list keyword or the end of the list.
- A specifier for a ***rest*** parameter. The lambda-list keyword &rest, if present, must be followed by a single *rest* parameter specifier, which in turn must be followed by another lambda-list keyword or the end of the lambda-list.
- Specifiers for ***keyword*** parameters. If the lambda-list keyword &key is present, all specifiers up to the next lambda-list keyword or the end of the list are *keyword* parameter specifiers. The keyword parameter specifiers may optionally be followed by the lambda-list keyword &allow-other-keys.
- Specifiers for ***aux*** variables. These are not really parameters. If the lambda-list keyword &key is present, all specifiers after it are *auxiliary variable* specifiers.

"

Reference: http://www.cs.cmu.edu/Groups/AI/html/cltl/clm/node64.html