Programming in Lisp

Lecture #4
Kenneth W. Flynn
RPI CS

Outline

- Items from last time
- Control
  - Iteration
- Functions
- Questions (Homework, Exam)

Exam 1

- Exam #1 will be
  - 9/22 6-8 pm (Location TBA)
  - 9/25 4-6 pm (Location TBA)
- Students who cannot make the first exam time should email me today. Only students who have notified me in advance will be permitted to take the exam on 9/25!

Items From Last Time

- When accessing members in a structure, the access function is of the form:
  - (defstruct rectangle width length)
  - (setf x (make-rectangle))
  - (rectangle-width x)
  - (rectangle-length x)
  - INCORRECT: (x-length)

Items From Last Time II

- 0 is not false

```
> (if 0 (format t "~%0 is not
  false!~%") (format t "~%0 is
  false!~%"))
  0 is not false!

  NIL
```

Control

- Iteration
- Conditionals
- Multiple Values
- A Note On Scope

Iteration: Do

- do
  - (do ((Variable initial-binding update-expression)
    (Variable initial-binding ...) ...) ;Variables
    ((ending-predicate) return-value) ;Returns
    (expression) ...
)
- Also do* (evaluates bindings in order each time)

Do, a loop, a useful loop...

- Order
  - Initial values are bound
  - Loop condition is checked (if reached, return)
  - Evaluate expressions
  - Update variables
  - Check loop condition...

Do examples! Now!

```
> (do* ((num 9 (- num 1))
        (root (sqrt num) (sqrt num))
        (lst (cons root 'nil)
             (cons root lst))
       )
       ((= num 1) lst)
       ()
  )
(1.0 1.4142135623731 1.73205080756888
 2.0 2.23606797749979 2.44948974278318
 2.64575131106459 2.82842712474619 ...)
```

Iteration Also

- *dolist*
  - Iterates through list items
- dotimes
  - Your basic for loop
- If you understand *do*, you can follow these.
- Refer to p. 88 of Graham for gory details...

Conditionals

- (cond ((predicate) (expressions))
  ((predicate) (expressions)) ...
)
- Powerful! Replaces if then else if then else ...

```
> (cond ((and t nil) 'Nope)
        ((or nil nil) 'Still-nope)
        ((or 13 (/ 1 0)) 'Ah-ha!)
        (t 'Default)
  )
AH-HA
```

Multiple Values

- For functions that return multiple values, use *(multiple-values-bind)*
- By example:
  - `> (multiple-value-bind (x pos) (read-from-string "123") (format t "~%Read the number: ~A up to position: ~A~%" x pos))`
  - `Read the number: 123 up to position: 3`

A Note On Scope

- *let*, *defun* both create a new *lexical context*
- Scope!
- Local variables overide globals, just like in C
- This issue is somewhat more complicated then we will cover

Functions

- Functions created with (defun) are global
- Local functions can be created with (labels)
  - Similiar to let
  - Instead of variables and bindings, include function definitions

labels Example

```
(defun silly (x)
  (labels ((add1 (x) (+ 1 x))
           (add2 (x) (+ 2 x)))
    (add2 (add1 x))
  )
)
```

Parameters

- Function parameters can be of 3 types
  - Required
    - Calls must include these parameters
  - Optional
    - Default to nil (or specified value) if not present
  - Key
    - Default to nil (or specified value) if not present; passed using :key syntax

Optional Parameters

```
&optional
(defun add (x &optional (y 0))
  (+ x y)
)
ADD
> (add 2)
2
> (add 2 1)
3
```

Keyword Parameters

```
&key
> (defun our-cons
          (&key (left 'Blank)
                (right 'Blank))
   (cons left right)
  )
OUR-CONS
```

Keyword Examples

```
> (our-cons)
  (BLANK . BLANK)
> (our-cons :right 'Hello)
  (BLANK . HELLO)
> (our-cons 'Hello)
  ;; Error: Keyword without
> (our-cons :left 'Hello :right 'World)
  (HELLO . WORLD)
```

Rest Parameters

- + takes any number of arguments
- We can do this with `&rest`
- Any values after the rest token will be bound into a list, and that will be bound to the variable following the rest token
- Combining rest and keyword parameters does not do what seems intuitive

Rest Example (zzz...)

```
(defun our-adder (&rest args)
  (do* ((sum 0 (+ sum curnum))
        (curnum 0 (car nums-list))
        (nums-list args (cdr nums-list))
       )
       ((null nums-list) (+ sum curnum))
       ()
  )
 )
```

Closure

- Functions which reference variables defined outside of their own lexical context are said to be *closures*.
- Functions are defined in some lexical scope, even though they are part of the global environment.
- Variables from the scope in which they are defined "stick with them" when they are called from outside that scope.

Closure Example (Slam!)

```
> (let ((noise 'Slam))
    (defun slam () noise)
   )
  SLAM
> (slam)
  SLAM
> (setf noise 'Ding)
  DING
> (slam)
  SLAM
```

That's It!

- Question & Answer Time
  - Homework #1
  - Exam #1
- Exam #1 will cover Chapters 2-6