Adrian Velonis, Haley Nolan, and Maya Johnson
Principles of Programming Languages
Deepak Kumar
December 3, 2020

Assignment #5

Subroutines

Functions in Common Lisp can be created using the defun keyword, followed by the function name and any parameters being passed to the function, with the function body below. To call a function, one can use funcall() or apply().

```
(defun add(x, y)
    (+ x y))

(defun hello-world()
    (format t "Hello, world!"))

(defun calladd()
    (funcall add(x y)))

(defun calladd2()
    (apply add(x y)))
```

Functions can also be created using lambda expressions, without explicitly naming them. This is done using the `lambda` macro.

```
(lambda (x)
    (= 0 (mod x 2)))
```

This function uses the modulo operator to determine the parity of the value passed to x.

The syntax `#'` can be used to signify that the program is searching for a function name, rather than a value of the function. This may be done to call on anonymous/lambda functions.

```
(funcall #'add(x y))
(funcall #'add '(1 2))
(funcall #'(lambda (x y) (+ x y)) 2 3)
```

Parameter Passing

Common Lisp uses call by sharing to pass parameters. Variables are references to objects, so that reference is passed. Hence, the formal and actual parameters refer to the same thing.

Common Lisp allows for fixed and variable numbers of arguments in functions. For example, the built in "+" function takes a variable number of arguments. User-defined functions have a fixed number of arguments by default, but can be made to have a variable number. When this is done, there is one parameter designated to be a list of all the optional arguments.

To make user-defined function with variable number of arguments, list any required arguments first, and then "&rest" plus a name for the parameter list.

Format:

```
(defun func-name (required-parameters &rest args))
```

Example from [2]:

```
(defun count-arguments (&rest args)
      (length args))

(count-arguments 1 2 3 4 5)
-> 5
(count-arguments)
-> 0
```

The user can define keyword/named parameters. These parameters have a default value and are optional.

Format:

```
(defun func-name (&key (param-name1 default1) (param-name2
default2) …))
```

Call: `(func-name :param-name1 value1 :param-name2 value2)`

Example from [2]:

```
(defun poem (&key (rose-color 'red) (violet-color 'blue))
  (list 'roses 'are rose-color 'and 'violets 'are
violet-color))

(poem)
-> (ROSES ARE RED AND VIOLETS ARE BLUE)
(poem :rose-color 'pink)
-> (ROSES ARE PINK AND VIOLETS ARE BLUE)
(poem :violet-color 'violet :rose-color 'yellow)
-> (ROSES ARE YELLOW AND VIOLETS ARE VIOLET)
```

Error/Exception Handling

Defining Conditions

Common Lisp allows the user to define and initialize their own error conditions using `define-condition` and `make-condition`. The example below from [3] shows the definition of an error condition for dividing by 0 and its initialization for dividing 3 by 0.

```
(define-condition my-division-by-zero (error)
  (dividend :initarg :dividend
             :initform nil
             :reader dividend))

(make-condition 'my-division-by-zero :dividend 3)
;; #<MY-DIVISION-BY-ZERO {1005C18653}>
```

Once we’ve created this condition, we can use it in our error handling.

Ignoring Errors

`ignore-errors`: returns `NIL` and the error condition

```
(ignore-errors
  (/ 3 0))
; in: IGNORE-ERRORS (/ 3 0)
;     (/ 3 0)
;
; caught STYLE-WARNING:
;   Lisp error during constant folding:
;   arithmetic error DIVISION-BY-ZERO signalled
;   Operation was (/ 3 0).
;
; compilation unit finished
;   caught 1 STYLE-WARNING condition
NIL
#<DIVISION-BY-ZERO {1008FF5F13}>
```

Note that this is not the same error condition for dividing by 0 as we defined using `define-condition`; rather, this is a built-in error condition. This example is from [3].

Catching Errors

handler-case: can catch specified or unspecified error, similar to try/catch block in other programming languages

In this example from [3], there is no specified condition; if we catch any error, we return an error message and print the condition we caught using the call to values.

```
(handler-case (/ 3 0)
  (error (c)
    (format t "We caught a condition.~&")
    (values 0 c)))
```

Contrast the above example with the one from [3] below, where we specify the default condition for dividing by 0.

```
(handler-case (/ 3 0)
  (division-by-zero (c)
    (format t "Caught division by zero: ~a~%" c)))
```

In this example, we do not need to call values since we already know the error condition, and thus we can customize our error message. We can use handler-case in either of these ways, generally or specifically, to catch errors and give detailed messages to the user.

Throwing and Reporting Error Conditions

There are two main functions we can use to throw errors: error and warn. Each of these functions uses the lower-level function signal. The examples below (from [3]) show the use of error, but warn could be used instead.

error: throws condition and opens interactive debugger
warn: throws condition and does not open interactive debugger

```
(error "Error")
      ;;general error of type simple-error
(error 'my-error :message "This didn't work.")
      ;;custom error condition and message
```

We can also define custom error reports, as in the example code below from [3], which we would include in our definition of the my-division-by-zero condition.

```
(:report (lambda (condition stream)
     (format stream "You were going to divide ~a by zero.~&"
(dividend condition))
```

Using Assertions and Restarts

To include a check for something we know could throw an error, we can use `assert`. The example function below from [3] checks whether we're about to divide by 0 before dividing.

```
(defun divide (x y)
  (assert (not (zerop y)))
  (/ x y))
```

If we try to divide by 0 using this function, we get this:

```
(divide 3 0)
;; The assertion (NOT #1=(ZEROP Y)) failed with #1# = T.
;;    [Condition of type SIMPLE-ERROR]
;;
;; Restarts:
;;  0: [CONTINUE] Retry assertion.
;;  1: [RETRY] Retry SLIME REPL evaluation request.
;;  …
```

We can also include a list of changeable values after our assertion if we want to be able to try something else, as below:

```
(defun divide (x y)
  (assert (not (zerop y))
          (y)   ;; list of values that we can change.
          "Y can not be zero. Please change it") ;; new message
  (/ x y))

(divide 3 0)
;; Y can not be zero. Please change it
;;    [Condition of type SIMPLE-ERROR]
;;
;; Restarts:
;;  0: [CONTINUE] Retry assertion with new value for Y.
;;  1: [RETRY] Retry SLIME REPL evaluation request.
;;  …
```

Similarly, we can use `restart-case` to give more options for the restarts instead of just 0 and 1. This can also be used to request a new value for a variable within the restart.

```
(defun divide-with-restarts (x y)
  (restart-case (/ x y)
    (return-zero ()
      :report "Return 0"  ;; <-- added
      0)
    (divide-by-one ()
      :report "Divide by 1"
      (/ x 1))))

(divide-with-restarts 3 0)
;; Nicer restarts:
;;  0: [RETURN-ZERO] Return 0
;;  1: [DIVIDE-BY-ONE] Divide by 1
```

Handling Conditions

If we want to handle multiple possible conditions with specific functions for each of the conditions, we can use `handler-bind`. Within the handler, we include possible error conditions, what to do if that condition is thrown, and the code that might throw the condition. The example below from [3] is the general form of a `handler-bind` statement.

```
(handler-bind ((a-condition #'function-to-handle-it)
               (another-one #'another-function))
    (code that can...)
    (...error out))
```

This way, we can handle errors more specifically than as with a `handler-case`.

"Finally" Condition

As with other languages' try/catch/finally, we can simulate the "finally" case in Common Lisp using `unwind-protect`. This function will not stop the debugger from opening, but it will run the code within it regardless of whether an error condition is thrown (example from [3]).

```
(unwind-protect (/ 3 0)
  (format t "This is safe!~&"))
```

Data Abstraction

The Common Lisp Object System (CLOS) is what makes Common Lisp object-oriented. Lisp was developed before object-oriented programming and there were many experiments at ecorporating OOP in, of which CLOS is a synthesis of.

Common Lisp is class-based, so every object is an instance of a class. There is a root class T, which is the superclass of every other class. Common Lisp supports multiple inheritance, so one class can have multiple superclasses and is an instance of all of them.

Unlike most programming languages, Common Lisp doesn't use methods or member functions to associate behaviors with classes. Instead, it uses generic functions. Generic functions consist of a name and a parameter list, but no code and are defined with defgeneric (similar to defun, but without the body).

For example: `(defgeneric draw (shape)` where "shape" is a class
```
                (:documentation "draw the shape"))
```

From there, you use defmethod to define and implement the generic function. Methods have to have the same number of optional and required parameters as the generic function, although the names don't have to match.

For example:
```
(defmethod draw ((shape circle))
   ... )
```

Generic functions also have a feature name call-next-method. This is used at the end of method implementations of the generic function to call the next relevant method. If it is called with no arguments, it passes the current method's arguments.

For example:
```
(defmethod draw ((shape circle))
   ...
  (call-next-method))
```

Generic functions are usually associated with specific classes by specifying the class of the parameters (like "shape"). Since subclasses are instances of their superclasses, this allows for inheritance. If "triangle" and "circle" are both subclasses of "shape," both would be able to implement draw in the way described above.

Format of a class:
```
(defclass <class-name> (list of super classes)
  ((slot-1
     :slot-option slot-argument)
   (slot-2, etc))
  (:optional-class-option
   :another-optional-class-option))
```

These are all valid definitions of the person class. It doesn't inherit from anything, so the () after the class name are empty.

```
(defclass person ()
  ((name
    :initarg :name
    :accessor name)
   (lisper
    :initform nil
    :accessor lisper)))

(defclass person ()
  (name lisper))

(defclass person ()
  )
```

An example of a class with inheritance:

```
(defclass child (person)
  )
```

Instances of classes are created with make-instance, but it's good practice to define a constructor for it.

```
(defvar p1 (make-instance 'person :name "me" ))

(defun make-person (name &key lisper)
  (make-instance 'person :name name :lisper lisper))
```

Variables in class are accessible at any point outside the class. They're accessed using "slot-value"

```
(slot-value <object> <slot-name>)

(defvar p1 (make-instance ‘person :name “Bryn”))
(slot-value p1 ‘name)
-> “Bryn”

(setf (slot-value p1 ‘lisper “yes”))
(slot-value p1 ‘lisper)
-> “yes”
```

Resources

1. Lisp/Common Lisp mentions in the textbook's index
2. Understanding arguments in Common Lisp - ccrma, Stanford (parameter passing)
3. https://lispcookbook.github.io/cl-cookbook/error_handling.html (error handling)
4. http://cl-cookbook.sourceforge.net/functions.html (functions)
5. https://en.wikipedia.org/wiki/Defun (functions)
6. https://www.cs.cmu.edu/Groups/AI/html/cltl/clm/node81.html (functions)
7. http://www.gigamonkeys.com/book/functions.html (functions)
8. http://www.gigamonkeys.com/book/practical-a-simple-database.html (functions, etc.)
9. https://towardsdatascience.com/a-swift-introduction-to-common-lisp-16a2f154c423
10. https://lispcookbook.github.io/cl-cookbook/clos.html (classes)
11. http://www.gigamonkeys.com/book/object-reorientation-generic-functions.html (data abstraction)