CSCE 476/876 Lisp Tutorial #3

Former UTA: Samuel W. Flint
University of Nebraska–Lincoln

Fall 2023

Note: Put your code in `recitation3.lisp` to load your code with `(load "recitation3.lisp")` or `:ld "recitation3.lisp"`.

Today, we cover `do`, an advanced iteration construct, `defstruct`, and how to work on lists of structures.

1 Iteration with `do`

We have previously covered the most basic common iteration constructs, like `dolist`, `dotimes`, and more 'interesting' functions such as `mapcar` (and its variants) and `reduce`. Review the online documentation for more information.

The `do` loop (a macro) allows us to iterate over more than just one thing:

Listing 1: The `do` loop

```
(do ((var1 init1 step1)
     (var2 init2 step2)
     ...
     (varn initn stepn))
    (end-test . result)
  {declaration}*
  . tagbody)
```

To demonstrate how `do` works, we write a basic function `pair-elements-and-indices`. This function takes as input a list of items. It returns, as output, a list listing each item with with twice the value of its index. For example:
`(pair-elements-and-indices '(a b c d))`
should return:
`((a 0) (b 2) (c 4) (d 6))`.

Parts of the following code needs to be filled in, but the structure of `do` is provided:

Listing 2: Pair elements and indices

```
(defun pair-element-and-index (list)
  (let ((result nil))
    (do ((reduced-list list (cdr reduced-list))
         (index 0 (1+ index))
        ((endp reduced-list) (reverse result))
  ;; Fill me in
  )))
```

There is a variant of do, do*, which binds the iteration variables in sequence, meaning that each iteration variable can be used in the definition of a subsequent variable. Consider using do* to convert the graph representation used in "Second Steps In Lisp" to a proper adjacency list. To do this, you may need to use filter-if and remove-duplicates.

2 Building Structures

In C, we have structs, in Java, we have classes. In Common Lisp, we have both and structures are usually 'lighter' than classes. We use defstruct to define structures.

Let us build a person structure to store various details about individuals. We want to keep track of first name, last name, and age. Are there any other pertinent fields to consider?

Listing 3: Defining person structure

```
(defstruct person ; name, can also be (name options*)
  ; Slot (field) Definitions
  ; (name default-value options*)
  (first-name "" :type string)
  (last-name "" :type string)
  (age 0 :type integer)
  ; Any others?
  )
```

We create a new person using (make-person :first-name first-name :last-name last-name :age age).

Now, we can access the slots using. person-*slot-name*

(This style of argument passing is referred to as using "keyword arguments". If you do *not* pass a keyword, it will default based on the function's definition.)

2.1 Working with Lists of Structures

Now that we have a definition of a person structure, let us write a function to find a specific person in a list of people. Start by defininîng a list of people, like this:

Listing 4: Defining a list of people

```
(defvar *friends*
  (list
    (make-person :last-name "Smith"
                 :first-name "John"
                 :age 24)
    (make-person :last-name "Spence"
                 :first-name "Angela"
                 :age 27)
    (make-person :last-name "Johnson"
                 :first-name "John"
                 :age 10)
    (make-person :last-name "Cuevas"
                 :first-name "Jerome"
                 :age 37)
    (make-person :last-name "West"
                 :first-name "Dwayne"
                 :age 5)
    (make-person :last-name "Yoder"
                 :first-name "Keshawn"
                 :age 39)
    (make-person :last-name "Randolph"
                 :first-name "Salma"
                 :age 40)
    (make-person :last-name "Mayo"
                 :first-name "Stanley"
                 :age 95)
    (make-person :last-name "Parker"
                 :first-name "Ezekiel"
                 :age 101)))
```

If we type `*friends*` in the Lisp listener now, we will see a list of somewhat hard to read objects. This formatting can be remedied by redefining the print function of the structure, which we can do by replacing the `person` in the `defstruct` with:

```
(person
 (:print-function
  (lambda (object stream ignore)
    (format stream "<Person␣~a␣~a,␣~a>"
            (person-first-name object)
            (person-last-name object)
            (person-age object)))))
```

Now, let us find our friend Parker in the list. We can do this relatively easily using `find-if`, which takes a predicate and a list of things to find. This function will find the *first* item in the list that satisfies the predicate. So, to find him, use `find-if` and a `lambda` that takes a person and compares their first name with `string=`.

Now, we want to sort the list of friends to be ordered by age. There are two ways to do this, both use `sort`. The first uses a custom predicate, the second uses a standard predicate and provides a "key" function to get the sort key.

Listing 5: Sort with custom predicate

```
(sort *friends* #'(lambda (x y) (< (person-age x) (person-age y))))
```

Alert: sort is destructive in that, while it returns a sorted list, the original list may be altered. Thus, we usually do the following:

Listing 6: Updating the input list with the result of the sorting

```
(setf *friends*
  (sort *friends* #'(lambda (x y) (< (person-age x) (person-age y)))))
```

This second version, sorting by first name, is shorter and improves readibility:

Listing 7: Sort by specifying key function

```
(setf *friends* (sort *friends* #'string< :key #'person-first-name))
```

Usually, we read the list of friends from a file, like an address book, listing all our friends. Write a function that reads from a file a list of people and their age and, for each person, it creates the appropriate structure and 'pushes' its to the global variable `*friends*`:

Listing 8: Names to store in a file

```
(John Smith 24)
(Angela Spence 27)
(John Johnson 10)
(Jerome Cuevas 37)
(Dwayne West 5)
(Keshawn Yoder 39)
(Salma Randolph 40)
(Stanley Mayo 95)
(Ezekiel Parker 101)
```

3 The "Farmer's Dilemma"

Review and discuss the code of the "Farmer's Dilemma" on course website.