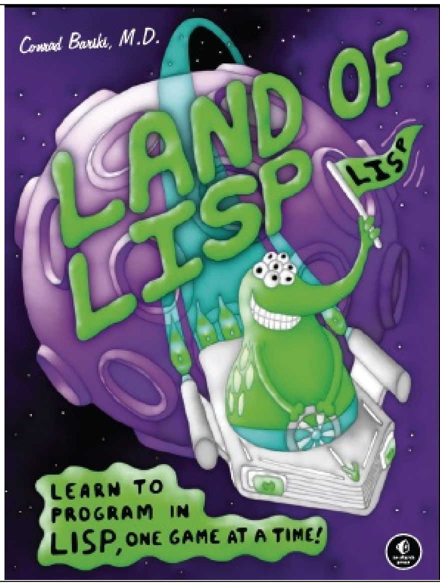


Ch. 9

1

2

Ch. 9 - Advanced Data Types and Generic Programming

- Arrays vs. Lists
- Hash Tables, similar to Alists
  - Working with Hash Tables
  - Returning Multiple Values
  - Hash table performance
- Arrays and hash tables may make your code a lot faster (use `time` command to check)
- Generic functions can be used to work with multiple data types.
- Sequence functions work on arrays, lists and strings.
- Orc Battle if you dare!

3

What are Data Structures For?

- Standard operations on Data Structures

- CREATE a new DS
- GET - look at a value in (an existing DS)
- SET - change or delete a value in
- DESTROY – get rid of a DS

- In particular, getters and setters are the most used operations in DSs!

4

Arrays: `makearray` and `aref`

```
                        ; creating an array
> (defparameter x
                (make-array 3)) ; create new array
#(NIL NIL NIL)          ; #( ..) identifies an array


> (aref x 1)      ; access an element with aref
NIL

; aref is a ‘getter’
; indices start at zero,
; thus x has indices 0, 1, and 2
```

5

Using a Generic Setter

- generic setter - the code for pulling a value out of a DS is identical to the code for putting data into that same DS - whether its an array, list, string, or something else!

- Using `setf` with a ‘getter’ function - lists

```
> (setf foo '(a b c))
(A B C)
> (second foo)
B
> (setf (second foo) 'z)
Z
> foo
(A Z C)
```

6

Generic Setter on Arrays

;; On arrays, use `aref` (getter) and

;; the generic setter `setf`

```
> (defparameter x (make-array 3))
#(NIL NIL NIL)

> (setf (aref x 1) 'foo) ; change position #1
FOO

> x
#(NIL FOO NIL)    ; position 1 is the second spot

> (aref x 1)
FOO
```

Speed - Arrays vs. Lists

- The function **nth** on a list is like **aref** on an array

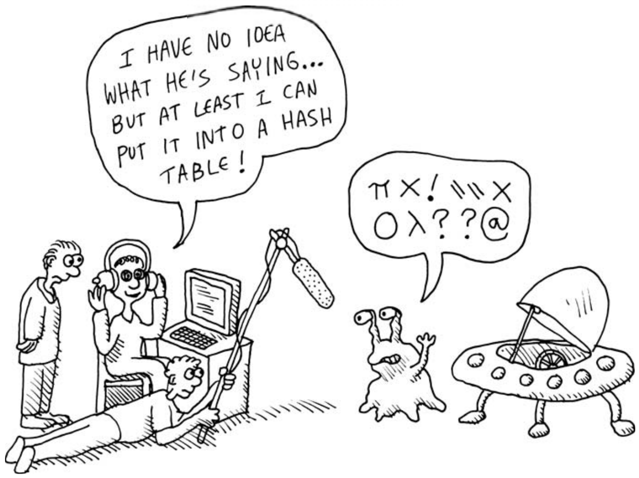
```
> (nth 1 '(foo bar baz))
BAR
```

- The main advantage of arrays is that they require only a constant amount of time to access any element.
- This is because Arrays are stored in contiguous memory (adjacent memory cells).
- In contrast, lists are strings of cons cells. To reach the nth item, one must follow n links (`cdr`s), so time is proportional to the size of n.

The Power of `setf` and Generalized References

```
> (setf foo (make-array 4))
#(NIL NIL NIL NIL)                          ; 4 elements, all default to nil

> (setf (aref foo 2) '(x y z))              ; Put a list in an array!
(X Y Z)

> foo
 #(NIL NIL (X Y Z) NIL)                     ; the result

> (setf (car (aref foo 2))
           (make-hash-table))               ; put a hash table in a list!
#S(HASH-TABLE)

> (setf (gethash 'zoink                     ; add to hash table
               (car (aref foo 2))) 5)
5
> foo                                       ; the final result
#(NIL NIL ( #S(HASH-TABLE (ZOINK . 5)) Y Z) NIL)
```


Hash Tables



Hash Tables

- Hash Tables are sort of like alists, except that they allow you to *access arbitrary elements more quickly*.
- Hash tables are so efficient that they can, at times, seem like **magic**!
- Almost all modern languages now offer the hash table data type

Using Hash Tables

```
> (defparameter x (make-hash-table))
#S(HASH-TABLE ...)

> (gethash 'yup x)
NIL;        ; nothing when searching for yup
NIL         ; second return value – successful gethash?

> (setf (gethash 'yup x) '25)
25

> (gethash 'yup x)
25 ;
T
```


Hash Table Performance

- Accessing and modifying a value inside a hash table requires only a constant amount of time, no matter how many items it contains.
- Suppose we have a hash table with only 10 items. We find it takes on average 1 millisecond to retrieve a value with its key.
- Now suppose that the hash table has 1,000,000 items in it. We still expect it to take only about 1 millisecond to retrieve a value!
- In other words, *no matter how big the table is, we can access items at a constant time of 1 millisecond.*

Hash Table Considerations

- Hash tables are indispensable when dealing with large amounts of data.
- Virtual memory paging may slow down access when tables are large.
- Hash collisions increase when tables get full.
- Inefficiency with small tables that need to grow to add many items.
- If there are only a few items to be stored and retrieved, then `alists` may be a better choice. Why??

Returning Multiple Values

```
;; Like get-hash, ROUND returns two values:
;; the integer divisor (1st) and remainder (2nd) :
> (round 2.4)
2 ;                         ; 1st return value
0                           ; 2nd return value
> (defun foo ()
      (values 3 7))         ; return 2 values
FOO                         ; result of defun
> (foo)
3 ;                         ; 1st return value
7                           ; 2nd return value
  ;; Recovering multiple values
> (multiple-value-bind (a b) (foo)
             (* a b))       ; bind values to args
21                          ; return last item evaluated
```


Common Lisp Structures

```
;; Create a new structure w/ name and slots
> (defstruct person           ; structure name
             name             ; first slot name
             age              ; next slot name
             waist-size       ; etc…
             favorite-color)
PERSON      ; DEFSTRUCT returns structure name
```


Working With Structures

```
> (defparameter *bob*          ; create a new structure
      (make-person       :name              "Bob"
                         :age               35
                         :waist-size        32
                         :favorite-color    "blue"))
*BOB*
      ; show printed representation of structures
> *bob*
#S(PERSON :NAME "Bob" :AGE 35 :WAIST-SIZE 32
:FAVORITE-COLOR "blue")

> (person-age *bob*)          ; READ/ACCESS slot value
35

> (setf (person-age *bob*) 36)    ; CHANGE slot value
36
```


Creating Structures Another Way

The Lisp reader can also create a person directly from the printed representation of the person, another great example of the print/read symmetry in Lisp:

```
> (defparameter *that-guy* #S(person :name "Bob" :age
35 :waist-size 32 :favorite-color "blue"))

> (person-age *that-guy*)
35
```


Handling Data in a Generic Way

- We wish to write **code that works with many types of data** — including **built-in** as well as **custom types** that we might create with `defstruct` — without superfluous repetition in our code.
- The easiest way to do this is to leave the type-checking work to someone else.

The same **sequence functions** work on lists, arrays, and strings! Example:

```
> (length '(a b c))
3
> (length "blub")
4
> (length (make-array 5))
5
```

Working With Sequences

```
> (defparameter *bob*          ; create a new structure
      (make-person      :name            "Bob"
                        :age             35
                        :waist-size      32
                        :favorite-color  "blue"))
*BOB*
      ; show printed representation of structures
> *bob*
#S(PERSON :NAME "Bob" :AGE 35 :WAIST-SIZE 32
:FAVORITE-COLOR "blue")

> (person-age *bob*)          ; READ/ACCESS slot value
35

> (setf (person-age *bob*) 36)    ; CHANGE slot value
36
```


Sequence Functions

Several sequence functions use a predicate to “look for” items in a sequence. The same functions work on all sequences.

```
> (find-if #'numberp '(a b 5 d))
5
> (count #\s "mississippi")
4
> (position #\4 "2kewl4skewl")
5
> (some #'numberp '(a b 5 d))
T
> (every #'numberp '(a b 5 d))
NIL
```


The REDUCE Function

```
> (reduce #'+ '(3 4 6 5 2))
20
      ; a larger example
> (reduce
      (lambda (best item)     ; anonymous function!
            (if (and (evenp item) (> item best))
                  item
                  best))
            '(7 4 6 5 2)      ; parameter to lambda
            :initial-value 0) ; keyword parameter
6                              ; initial value to best
```


Generic Functions vs. Generic Soap?

Mapping Functions on Sequences

```
;;; REDUCE gives the function in its 1st
;;; parameter successive items in the 2nd
;;; parameter to produce its result
> (reduce #'+ '(3 4 6 5 2))
20
          ;;; a larger example
;;; find the largest even number in a list
> (reduce  ; anonymous function definition
     (lambda (best item)   ; takes 2 parameters
           (if (and (evenp item) (> item best))
                item       ; return new best
                best))     ; return old best
           '(7 4 6 5 2)    ; parameter to lambda
           :initial-value 0); keyword init value
6
```


More Mapping Functions

```
;;; SUBSEQ returns the subsection between
;;; the numbers in its 2nd and 3rd parameters

> (subseq "america" 2 6)
"eric"

;;; SORT uses a predicate (2nd parameter)
;;; to arrange the items in its 1st parameter
;;; in the specified order

> (sort '(5 8 2 4 9 3 6) #'<)
(2 3 4 5 6 8 9)
```

Creating Your Own Generic Functions with Type Predicates

Common Lisp, like virtually all other Lisps, is a dynamically typed language. This means that **parameters or variables** in your code **can hold any type of data** — symbols, strings, numbers, functions, or whatever else you want to place in them.

The type predicates you will probably use most frequently are `arrayp, characterp, consp, functionp, hash-table-p, listp, stringp, and symbolp.`

A Generic Add Function

```
> (defun add (a b)      ; define function
      (cond
            ((and (numberp a) ; work with numbers
                  (numberp b))
             (+ a b))
            ((and (listp a)   ; work with lists
                  (listp b))
             (append a b))))
ADD
> (add 3 4)
7
> (add '(a b) '(c d))
(A B C D)
```


Supporting Multiple Data Types

- CL's `defmethod` lets us define **multiple versions of a function** that each **support different types**.
- Lisp checks the argument types at the time of the call and *chooses the correct version of the function automatically*.
- Type dispatching - having a compiler/ interpreter choose among different versions of a function based on argument types.

Generics Using DEFMETHOD

```
> (defmethod add ((a number) (b number))
      (+ a b))
ADD
> (defmethod add ((a list) (b list))
      (append a b))
ADD
; the same results as if using type predicates
> (add 3 4)
7
> (add '(a b) '(c d))
(A B C D)
```

`defmethod` is like `defun`, except that it allows us to *write multiple functions with the same name*. When using `defmethod`, we can explicitly state the type of each parameter in the function's argument list so that Lisp can use these type declarations to figure out the correct version of `add` for each situation.

Summary

- Arrays vs. Lists
- Hash Tables, similar to Alists
  - Working with Hash Tables
  - Returning Multiple Values
  - Hash table performance
- Arrays and hash tables may make your code a lot faster (use `time` command to check)
- Generic functions can be used to work with multiple data types.
- Sequence functions work on arrays, lists and strings.
- Orc Battle if you dare!