LISP CHEAT SHEET

A cheat sheet on some Lisp data types that are probably familiar to you as a non-Lisp coder.

COMMON NON-BASIC DATA TYPES

STRUCTURE	A structure (like a C struct)
CLASS AND INSTANCE	Objects with multiple inheritance. We'll not consider object methods.
HASH-TABLE	A hash table
LIST	A singly linked list
ARRAY	A multidimensional array, possibly resizable
VECTOR	A one-dimensional array, possibly resizable
SIMPLE-VECTOR	A one-dimensional array, not resizable
STRING	Generally a one-dimensional array of characters, not resizable

SEQUENCES: Super type of all lists and vectors (including strings)

HOW TO SET VALUES:

Use SETF on any expression which gets a value. For example, to get a value out of the hash table MY-TABLE, keyed with the key "foo", you say:

```
(gethash "foo" my-table)
```

Likewise, to set the value of keyed with "foo" on MY-TABLE to 3, you say:

```
(setf (gethash "foo" my-table) 3)
```

Let's try again. Let's say you have a list of sublists. To get the first element out of the fourth sublist in a list MY-LIST, you could say:

```
(first (elt my-list 3))
```

Likewise, to set that value to "Hello", you could say:

```
(setf (first (elt my-list 3)) "Hello")
```

Always use SETF. Do NOT use SETQ, SET, RPLCA, etc.

ALL SEQUENCES:

Making:
- make-sequence (rarely used)

Nth Element:
- elt

Size:
- length

Joining:
- concatenate

```
Copying:
        copy-seq
Subsequences:
        subseq
        search
        mismatch
Discovery:
        position             (see also: position-if, position-if-not)
        count                (see also: count-if, count-if-not)
        find                 (see also: find-if, find-if-not)
        member               (see also: member-if, member-if-not)
Type Testing:
        (typep MY-SEQUENCE 'sequence)
Element Testing:
        some
        every
        notany
        notevery
Bulk Operations:
        remove               (destructive variant: delete)
                             (see also: remove-if, remove-if-not)
                             (see also: delete-if, delete-if-not)
        remove-duplicates    (destructive variant: delete-duplicates)
        substitute           (destructive variant: nsubstitute)
                             (see also: substitute-if, substitute-if-not
                             (see also: nsubstitute-if, nsubstitute-if-not)
        fill
        reverse              (destructive variant: nreverse)
        sort                 (note sort is destructive)
                             (see also: stable-sort)
Iterating:
        map
        (lists have their own special iterators)
        (see also:    do )
        (see also:    loop )




LISTS: (remember, lists are sequences)

Literals:
        '(1 2 3 4 5)
        '()
        ()                   (there's no need for the quote)
        nil
Making:
        (list 1 2 3 4 5)
        (cons 1 (cons 2 (cons 3 (cons 4 (cons 5 nil)))))
Size:
        list-length          (this is slower than LENGTH, but can handle
                              circular lists.  Ordinarily you'd use LENGTH)
Nth element:
        nth                  (but you should use elt instead, except when
                              efficiency is paramount)
Getting:
        first
        rest
        last
        butlast
        nthcdr
        (also use sequence functions)
```

```
Joining:
        append              (destructive variant: nconc)
Bulk Operations:
        (use sequence functions)
Deep-copying:
        copy-tree
Type Testing:
        consp
        null
        atom                (not (consp LIST))
        listp               (or (consp LIST) (null LIST))
Iterating:
        dolist
        mapcar
        mapc
        (also use sequence functions)
        (far less common: mapl, maplist, mapcan, mapcon)




ARRAYS:

Literals:    #2A((1 2 3) (4 5 6))  ;; a 2x3 unextensible array

Making:
        (make-array '(2 3))        ;; there are many options here
Getting:
        aref
Dimensions:
        array-dimensions
Copying:
        From http://lemonodor.com/archives/000100.html

        (defun copy-array (array)
          (let ((dims (array-dimensions array)))
            (adjust-array
              (make-array dims :element-type (array-element-type array)
                           :displaced-to array)
              dims)))

Reshaping:
        adjust-array
Type Testing:
        arrayp  ;; note: also works for all vectors and strings
Iterating:
        For one-dimensional arrays, that is, vectors, use sequence functions.
        For n-dimensional arrays, you can 'cast' them into vectors like this:

            (let ((vec (make-array (apply #'* 1 (array-dimensions my-array))
                                   :displaced-to my-array)))
                ... now you can iterate over the vector 'vec'
            )




VECTORS:  (Extensible vectors.  Remember, vectors are sequences and also arrays)
```

```
Literals:
        None.  But simple-vectors have them.  You could define literals
        of the form #!(a b c) to create extensible, non-simple vectors
        with the following code:

        (set-dispatch-macro-character #\# #\!
                   (lambda (s c n)
                      (let ((v (read s)))
                          `(make-array ,(length v) :initial-contents ',v
                             :adjustable t :fill-pointer t))))
Making:
        (make-array '(10) :adjustable t :fill-pointer t)  ;; other options
Getting:
        (use sequence functions)
Copying:
        (use sequence functions)
Bulk Operations:
        (use sequence functions)
Reshaping:
        vector-push-extend      ;; note: don't use the variant vector-push
        vector-pop
Type Testing:
        vectorp  ;; note: also works for simple-vectors and strings
Iterating:
        (use sequence functions)
```

```
SIMPLE-VECTORS:   (Not extensible.  Remember, simple-vectors are sequences,
vectors, and arrays)

Literals:
        #(1 2 3 4 5)
Making:
        (vector 1 2 3 4 5)
Getting:
        (use sequence functions)
Bulk Operations:
        (use sequence functions)
Copying:
        (use sequence functions)
Type Testing:
        simple-vector-p
Iterating:
        (use sequence functions)
```

```
STRINGS:    (Not extensible.  Remember, strings are sequences,
vectors, and arrays)

Literals:
        "Hello, World!"
Making:
        (make-string 10 :initial-element #\a)
Getting:
        (use sequence functions)
```

```
Copying:
        (use-sequence functions)
Bulk Operations:
        string-upcase        (destructive variant: nstring-upcase)
        string-downcase      (destructive variant: nstring-downcase)
        string-capitalize    (destructive variant: nstring-capitalize)
        string-trim
        string-right-trim
        string-left-trim
Type Testing:
        stringp
Comparison:
        string=              (case insensitive: string-equal)
        string<              (case insensitive: string-lessp)
        string>              (case insensitive: string-greaterp)
        string<=             (case insensitive: string-not-greaterp)
        string>=             (case insensitive: string-not-lessp)
        string/=             (case insensitive: string-not-equal)
Iterating:
        (use sequence functions)

;; Note: A string is just a fixed-length a vector of characters.  "string"
;; is not a formal data type so much as a convention.  The actual data type
;; is typically SIMPLE-STRING, through it'll show up in TYPEP as something
;; like (SIMPLE-ARRAY CHARACTER (5)).
;;
;; As a result, in fact you *can* in fact make extensible strings, by simply
;; making adjustable VECTORS of the type CHARACTER.  These will in fact be
;; treated as if they were simple strings, but you can tack stuff onto them
;; by calling VECTOR-PUSH-EXTEND (or remove stuff with VECTOR-POP).  This is
;; a quite uncommon thing to do.  To make an extensible string, you could
;; say something like:

        (make-array '(10) :initial-element #\a :adjustable t
                          :fill-pointer t :element-type 'character)



HASH TABLES:

Literals:
        None, but inspired by the following reddit comment:
        http://www.reddit.com/r/programming/comments/7sfro/response_to_problems_with_lisp/

        ... the following code will let you say
        #?((key val) (key val) (key val))
        ... key and val can be anything, and should not be quoted.
            for example, #?((a b) (c 4) ("hi" (one two)) (#(1 2 3) #\a))

        (set-dispatch-macro-character
          #\# #\?
          (lambda (s c n)
            (declare (ignore c))
            (declare (ignore n))
            (let ((v (read s)) bag)
              (dolist (x v)
                (push `(setf (gethash ',(first x) hash) ',(second x)) bag))
              `(let ((hash (make-hash-table :test #'equal))) ,@bag hash))))


Making:
        (make-hash-table :test #'equal)  ;; I suggest using #'equal always
Getting:
        gethash
        hash-table-count
```

```
Deleting:
        remhash
Copying:
        The following is a somewhat modified version of the COPY-HASH-TABLE
        function from the "hash-tables.lisp" file of the "Alexandria" project.

        (defun copy-hash-table (table &key (key-copy #'identity)
                                           (value-copy #'identity))
          "Returns a copy of hash table TABLE, with the same keys and values
          as the TABLE. Before each of the original key/value pairs are set into
          the new hash-table, KEY-COPY is invoked on the key and VALUE-COPY is
          invoked on the value.  As both of these default to #'IDENTITY, a
          shallow copy is returned by default."

          (let ((copy (make-hash-table
                       :test (hash-table-test table)
                       :size (hash-table-size table)
                       :rehash-size (hash-table-rehash-size table)
                       :rehash-threshold (hash-table-rehash-threshold table))))
            (maphash (lambda (k v)
                       (setf (gethash (funcall key-copy k) copy)
                             (funcall value-copy v)))
                     table)
            copy))

Bulk Operations:
        clrhash
Type Testing:
        hash-table-p
Iterating:
        maphash




STRUCTURES:
        (We will imagine you're making a structure called FOO, with three
         fields, BAR, BAZ, and QUUX).

Declaring:
        (defstruct foo (bar 4) (baz "yo") quux)

        ;; This makes a struct data type called FOO with the three fields, where
        ;; BAR is set  by default to 4, BAZ is set by default to "yo", and QUUX
        ;; is set by default to NIL
        ;;
        ;; Automatically created are several functions: MAKE-FOO, FOO-BAR,
        ;; FOO-BAZ, FOO-QUUX, FOO-P (among others), as discussed below.
        ;;
        ;; You MUST declare a FOO type before you can create one
        ;; There are many other options to DEFSTRUCT -- look them up.
        ;; Particularly useful are options for documentation and for defining
        ;; how the structure is printed to the screen.

Literals:
        #S(FOO :bar 17 :quux "hey")

        ;; this makes a FOO with BAR set to 17, QUUX set to "hey", and BAZ set
        ;; to its declared default ("yo")

Making:
        (make-foo :bar 17)

        ;; this makes a FOO with its BAR set to 17, and its BAZ and QUUX set to
```

```
        ;; the default ("yo" and nil)

Getting:
        (foo-bar --the-object-- )    ;; gets the BAR slot of the object
        (foo-baz --the-object-- )    ;; gets the BAZ slot of the object
        (foo-quux --the-object-- )    ;; gets the QUUX slot of the object

        ;; setting is as normal:  (setf (foo-bar --the-object--) 17)

Copying:
        copy-structure

Type Testing:
        (foo-p --the-object-- )
        (typep --the-object-- 'foo)




CLASSES AND INSTANCES:
        (We only cover how to make classes and instances, not how to define
        methods attached to them.  So basically we're treating objects here
        as if they were structures.)

        (We will imagine you're making a class called FOO, with three fields,
         BAR, BAZ, and QUUX.  Furthermore, you are making a class called XYZZY
         which is a subclass of FOO and has an additional field called PLUGH.

Declaring:
        (defclass foo ()
              (bar
               (baz :initform 4 :reader get-baz :type fixnum)
               (quux :initarg :quux :accessor get-quux :allocation :class))
              (:documentation "FOO is a very nifty class"))

        ;; This makes a class data type called FOO with the three fields, BAR,
        ;; BAZ, and QUUX. The documentation for the FOO class is specified.
        ;;
        ;; BAR is UNINITIALIZED.
        ;; BAZ is initialized to 4 and is restricted to be of the type FIXNUM
        ;;     (integers).
        ;; QUUX is initialized by a keyword parameter passed at
        ;; construction-time, called :quux. if this parameter is not provided,
        ;; then its initial value is nil.
        ;;
        ;; BAR and BAZ are instance variables, but QUUX is a static (class)
        ;; variable, to use Java paralance.  That is, each FOO instance has its
        ;; own BAR and BAZ, but QUUX is a single variable shared by all FOO
        ;; instances.
        ;;
        ;; Furthermore, BAR cannot be accessed at all except through SLOT-VALUE.
        ;; BAZ can be read using a function called GET-BAZ, but cannot be
        ;; written to except through SLOT-VALUE. QUUX can be read via GET-QUUX,
        ;; and written via (SETF (GET-QUUX ... ))


        (defclass xyzzy (foo)
              ((plugh :accessor whatever :documentation "PLUGH is important!")))

        ;; This makes a class data type called XYZZY with a single field, PLUGH.
        ;; XYZZY is a subclass of foo.
        ;; PLUGH is UNINITIALIZED
        ;; The accessor function for PLUGH will be called WHATEVER.
        ;; The documentation for PLUGH is provided.
```

```
Literals:
        ;; there are no literals for objects

Making:
        (make-instance 'foo :quux 997.0)
        (make-instance 'xyzzy :quux 972.1)

Resetting:
        reinitialize-instance

Getting:
        ;; note in all cases below, if the slot is uninitialized, then the
        ;; method SLOT-UNBOUND is called, and by default it thows a condition
        ;; (an exception) rather than return something like NIL. If you'd like
        ;; to avoid this, you could either define :initform for every single slot,
        ;; or you could override the behavior of the SLOT-UNBOUND method to return
        ;; NIL for unbound slots in objects of class FOO like this:
        ;;
        ;; (defmethod slot-unbound (cls (obj foo) slot) nil)
        ;;
        ;; Alternatively for ALL objects of ALL classes:
        ;;
        ;; (defmethod slot-unbound (cls obj slot) nil)

        (slot-value 'bar --the-instance--)  ;; gets the BAR slot of the object
        (whatever --the-instance-- )        ;; gets the PLUGH slot of the object
        (get-quux --the-instance-- )        ;; gets the QUUX slot of the object

        ;; setting is as normal:
        (setf (slot-value 'bar --the-instance--) 17)
        (setf (whatever --the-instance--) "hello")

        ;; Testing for whether a slot is bound:
        (slot-boundp --the-instance-- 'bar)
        (slot-boundp --the-instance-- 'quux)

        ;; You can also access slots or accessors as if they were local variables
        ;; using the (not particularly efficient) macros

        with-slots
        with-accessors

Copying:
        ;; None built in, because objects can get arbitrarily complex with
        ;; nonstandard "metaobject" protocols (MOPs).  It's not a good reason.
        ;; But you might look at
        ;; https://stackoverflow.com/questions/11067899/is-there-a-generic-method-for-
cloning-clos-objects

Comparison:
        ;; None built in, for the same basic reason as copying

Type Testing:
        (foo-p --the-instance-- )
        (typep --the-instance-- 'foo)

Custom Printing:
        (defmethod print-object ((obj foo) stream) ... )
```