Function Standards

This list describes the most commonly used Common Lisp and GDL functions. It also describes *how* to use the functions within the KE-works *standards*. When special attention is needed for these standards, the text is underlined.

For explanation of the functions you can try several *references*:

- Use C-h f in Emacs to access a function description (CL functions only)
- Use the GDL YADD (for GDL functions)
- Check the Common Lisp Hyperspec by searching Google with `clhs <function>`
- Use the book *ANSI Common Lisp* by Graham, available at office or at this link.

Table of Contents

Function Standards
- Setting, declaring & defining variables & functions
- Logical operators
- Conditionals
- List operators
- Mapping operators
- Iterations
- Mathematical operators
- String operators
- Robustness
- I/O
- System Operators
- System
- Others
- GDL functions

Code Conventions
- Naming standards
- Annotations
- Error & warning messages

Setting, declaring & defining variables & functions

- `setq`, `setf`: Discussion on the difference of `setq` and `setf`: http://stackoverflow.com/q/869529/1097797
- `let*`, `let`: The function `let*` and `let` makes you define a local variable. `let*` allows for interdependencies of variables within the same `let*` expression. Therefore `let*` should be used by default.

```
(let* ((area 90)
       (span 30)
       (aspect-ratio (/ (expt span 2) area))
      ...)
```

```
;; while
(let ((area 90)
      (span 30))
  (let ((aspect-ration (/ (expt span 2) area))
        ...))
```

- `defun`: Standard Common Lisp function to define functions.

```
(defun my-function (arg-1 &optional (opt-3 default-3)
                          &key (key-5 default-5) key-6
                          &rest args)
       ... )
```

- `declaim`, `declare`: These functions declare a variable and its type. `declaim` is used for global variables and `declare` for local variables. In Common Lisp you are no mandatory to declare a variables type, but in a later stage of the program it can reduce calculation time and memory as less space is being reserved.

```
(declaim (fixnum *global*))
```

- `defparameter`, `defvar`, `defconstant`: These are functions to define global parameters. A constant defined through `defconstant` cannot be changed later on. The most important difference between `defvar` and `defparameter` is that `defvar` does not overwrite the original value when you use it twice on the same variable. Furthermore, with `defvar` value argumenter is optional, while it is required with `defparameter`. All

functions have an optional third argument of type string, that will be the documentation string for the YADD. Use `defconstant` for constants, `defparameter` when its value is already known. A documentation string is a KE-works standard!

```
(defvar *var* 2.3)
=> var
var
=> 2.3
(defvar *var* 3.3)
=> var
var
=> 2.3
```

- `boundp`: Checks whether the variable has an assigned value.
- `flet`, `labels`: Both functions let you define a local function, with the difference that within `label` you can use the local function within its own definition. Use `labels` by default, just like `let*` is standard practice.

Example from CLHS:

```
(flet ((dummy-function () 'shadow))
      (funcall #'dummy-function)) =>  SHADOW
```

```
defun recursive-times (k n)
   (labels ((temp (n)
              (if (zerop n) 0 (+ k (temp (1- n))))))
     (temp n))) =>  RECURSIVE-TIMES
 (recursive-times 2 3) =>  6
```

Logical operators

- `and`, `or`: These are typical boolean operators and therefore use them only on real booleans only, such that the correct data type is expected.

```
;; wrong
(and (sin x) (listp y))
;;correct
(and (listp y) (every #'oddp y))
```

- `not`: use `not` as the negation of an argument. Although it does the same thing as `null`, the argument of `not` may be expected to be a boolean.

```
(when (not (= 0 var)) (/ 1 var)) ;; normally one would use 'unless'
```

- `<`, `>`
- `eq`, `eql`, `equal`, `=`, `string-equal` : Many compare functions exist, here it is explained when and how to use them.
  - `eq` compares if the two objects point to the same memory location (identity). Cannot be used for numbers and characters! This function is faster than `eql`, so use `eq` by default when no numbers and characters are used.
  - `eql` is similar to `eq` and more robust than `eql` for number and character types, but therefore also slower. Use it in all cases where `eq` is not applicable.
  - `equal` compares if both arguments print to the same value. Preferably do not use it, as it may be mistaken with `eq` or `eql`.

```
(let* ((a "KE-works")
       (b "KE-works"))
  (list (eql a b)
```

```
        (equal a b)))
=> (nil t)
```

- `=` checks whether the difference between two arguments is zero. By default use it for number comparison.

```
(equal 1 1.0)
=> nil
```

```
(= 1 1.0)
=> t
```

- `string-equal` compares two strings, irrespective of its case.

```
(equal "ke-works" "KE-works")
=> nil
```

```
(string-equal "ke-works" "KE-works")
=> t
```

- type checking (`numberp`, `listp`, `stringp`, `typep`, etc.): You can check date types with these functions that have postfix `p`. The more generic function `typep` can be used for all types, also newly defined types. Please use the most concrete function (least abstract) by default. I.e. `typep` is not preferred.

```
;; too abstract
(typep 1.0 'number)
=> t
```

```
;; more concrete
(numberp 1.0)
=> t
```

Conditionals

- `when`, `unless`: use these in case of **single** branch conditionals

```
(when (numberp x) (print x))
```

- `if`: use this function in case of **double** branch conditionals

```
(if (numberp x) (cos x) nil)
```

- `cond`: use this function in case of **multi**-branch conditionals, where the `case` functions are not applicable. Specify the most expected condition on top as this will save calculation time.

```
(cond ((numberp x) (* 2 x))
      ((stringp x) (print x))
      ((listp x) (reverse x))
      (t (print "Not of type number, string or list"))
```

- `case`, `ccase`, `ecase`: The `case`-functions can select different branches by means of a keyword or symbol. Use them for discrete distinct cases. `case` returns `nil` if no keyword matches and allows you to use the `otherwise` clause. `ecase` and `ccase` do not allow you to use `otherwise` and return a non-correctable and correctable error respectively if the specified keyword does not match one of the cases. Use `ecase` by default to keep track of possible erroneous input or specify your own checks and error handling when

`case` seems more suitable. (Also see the section Robustness)

```
(let* ((month :april))
  (case month
    ((:january :march :may :july :august :october :december) 31)
    ((:april :june :september :november) 30)
    (:february 28)
    (otherwise "unknown month")))
=> 30
```

List operators

Creation

- `list`, `quote`, `'`: You can create list by using three functions. `list` is the most intuitive and clear function; use `list` by default in code files. As `quote` literally evaluates the argument, it can be used for automatically writing code. `'(` is exactly the same as `(quote`, `'(` is useful for short notation during testing and in training examples.

```
(list (list :key 'a) (list :type 'b))
=>((:key a) (:type b))
```

Access

- `car`, `cdr`, `caddr`, etc.: These function are very basic in Common Lisp. It is perfectly fine to use `car` and `cdr`, but do not use functions like `caddr` or variations as they are very unintuitive. Preferably use `first`, `second`, etc., `nth` or `subseq` instead.
- `first`, `second`, etc.: Very intuitive function to access certain elements in a list. By default use these, unless you do not know beforehand which position from the list you will need. Then use `nth`.

```
;; too abstract
(nth list 2)
;; as it is similar to
(third list)
```

- `nth`, `elt`: `nth` and `elt` access a certain position in a list and a sequence respectively. This means that `elt` can also be used on strings. Furthermore their argument order is mirrored. Both functions can be used, but as `nth` is commonly used more often, use `nth` by default. `nth` is also more robust by returning `nil` if the requested element does not exist.

```
(nth 2 '(a b c d))
=> c
```

```
(nth 5 '(a b c d))
=> nil
```

```
(elt "abcd" 2)
=> #\c
```

- `subseq`: A very useful function to access a part of a sequence (i.e. lists and strings).

```
(subseq '(a b c d) 1 3)
=> (b c)
```

- `lastcar`, `last`, `butlast`: The functions `last` and `butlast` return a list with the last element and all elements but the last of the input list. To directly access the last *element*, `lastcar` can be used, which is similar as `(car (last list))`. Mostly `lastcar`

is only necessary. `butlast` is useful for going up one path level.

```
(last '(a b c d))
=> (d)
```

```
(butlast '(a b c d)
=> (a b c)
```

```
(lastcar '(a b c d))
=> d
```

Expansion & reduction

- `append`, `nconc`:
- `union`, `intersection`, `set-difference`
- `cons`, `push`, `adjoin`: `cons`, `adjoin` and `push` all add an element in front of a list, but `push` is destructive to a list and `adjoin` testes whether the element is already part of the list. Use `push` within iterative functions to append a result list, for example:

```
(let* (result)
  (dolist (elem list result)
    (unless (zerop elem)
      (push (/ 1 elem) result))))
```

```
(setq lst '(1 2))
=> (1 2)
(cons 'a lst)
=> (a 1 2)
lst
=> (1 2)
```

```
(adjoin 'a '(b c))
=> (a b c)
(adjoin 'b '(a b c))
=> (a b c)
```

- `pop`: This function retrieves the first element from a list, but just like `push`, `pop` is destructive.

```
(setq lst '(1 2 3))
=> (1 2 3)
(pop lst)
=> 1
(pop lst)
=> 2
```

- `remove`, `delete`: These functions can remove certain elements from a list, with the difference that `delete` is destructive.
- `remove-duplicates`, `delete-duplicates`: These functions remove duplicate elements from a list, where `delete-...` is again destructive. Use a predicate to test every element.
- `remove-if`, `remove-if-not`, `delete-if`, `delete-if-not`: These functions can conditionally remove elements from a list, where `delete-...` is again destructive.

```
(remove-if #'oddp '(1 2 3 4 5 6))
=> (2 4 6)
```

Information

- `position`, `member`: both `position` and `member` are able to test whether an element is part of a list. `member` returns a *subsequence* of the list starting with the test element, while `position` returns the *position* of the element (N.B. zero-based) of the element within the list. Additionally `position` will also work with *strings*. When the test element is not part of the list (or string), both function will return `nil`. Both functions are *non-destructive*. Use `position` by *default* as it works on lists *and* strings and is more intuitive in its return value. Use `member` only if it will have significant benefits.

```
(member '2 '(1 2 3 4))
=> (2 3 4)
```

```
(member '5 '(1 2 3 4))
=> nil
```

```
(position '2 '(1 2 3 4))
=> 1
```

```
(position #\c "abcd")
=> 2
```

```
(position '5 '(1 2 3 4))
=> nil
```

- `length`: Returns the length of a sequence.

```
(length '(a b c))
=> 3
```

```
(length "abcd")
=> 4
```

- `count`

Manipulation

- `apply`, `reduce`: very useful functions to use the elements of a list as arguments to a certain function. `apply` actually uses all input list's elements as `&rest` arguments, while `reduce` makes subsequent calls to the function. Therefore use `reduce` by default as it is the faster one, unless its way of calling is not applicable.

```
(reduce #'fn '(a b c d))
;; is equivalent to
(fn (fn (fn 'a 'b) 'c) 'd)
;; while
(apply #'fn '(a b c d))
;; is equivalent to
(fn 'a 'b 'c 'd)
```

The below example shows how to find the maximum value in a list.

```
(reduce #'max list)
```

- `reverse`, `nreverse`: This function reverses the input list. `reverse` itself is non-destructive and hence uses more memory. Therefore use `nreverse` by default when you do not want to store the original list.

Checking

- `some`, `every`: Very useful functions to test the contents of a list and return a single boolean value. `notany` is the negation of `some`.

```
(some #'evenp '(1 2 3))
=> t
```

```
(every #'> '(1 3 5) '(0 2 4))
=> t
```

- `null`: use `null` to test for the *emptiness* of a list. Although it does the same thing as `not`, the argument may be expected to be of list type.

Mapping operators

- `mapcar`: use `mapcar` to perform an operation on every element of a list and return the answer in a list of equal length, for example *vector operations*. Make sure not to use too difficult lambda functions.

```
(mapcar #'< '(0 1 2) '(2.5 1.5 0.5))
=> (t t nil)
```

```
(mapcar #'(lambda (x) (expt x 3)) '(1 2 3))
=> (1 8 27)
```

  - `mapcan`: use `mapcan` similar to `mapcar` and to additionally splice together the values returned by the function, which must be of list type. Might be useful for creating property lists.

```
(mapcan #'list '(:volume :message :color)
               '(1.5 "Hello" :red)
=> (:volume 1.5 :message "Hello" :color :red)
```

Iterations

Iteration function can sometimes be quite similar to the mapping functions. The preferred application order of the functions is: `mapcar`, `dolist`, `dotimes` and `loop`. Only use the less preferred functions if they are more suitable (e.g. in understandability and efficiency).

- `dolist`: Use `dolist` when you want to iterate over a certain list and there is no return value or is a list of different length as the input length.

```
(dotimes (dir dirs)
   (unless (probe-file dir) (make-directory dir))
```

```
(let (result)
  (dotimes (elt lst (nreverse result))
    (unless (= elt 0) (push (/ 1 elt) result))))
```

- `dotimes`: Use `dotimes` when the number of iterations is not related to a list.
- `loop`: Loop is a very powerful macro for iterations. Its syntax is very different from default Common Lisp and it is hard to learn and understand. As the use of `loop` is controversial, it is preferred not to use it at all, unless it has significant advantages. Keep in mind that it must be *understandable* for others within KE-works as well. Below are some examples to make `loop` understandable.

```
(loop for x from 0 to 9
      do (princ x))
=> 0123456789
```

```
(loop for x in '(1 2 3 4)
      collect (1+ x))
=> (2 3 4 5)
```

More examples of `loop` can be found on page 240-244 of Graham.

Mathematical operators

- `+`, `-`, `*`, `/`, `expt`, `sqrt`, `1+`, `1-`: Useful mathematical operations which should be known by heart.
- `floor`, `truncate`, `round`, `ceiling`, etc.: Functions for number operations. `floor` truncates towards negative infinity, `truncate` truncates towards zero, `ceiling` truncates towards positive infinity and `round` rounds towards the nearest integer for the quotient.
- `mod`, `rem`: return the remainder of the `floor` and `truncate` operation respectively.

String operators

- `string`
- `string-append`: Appends two strings together into one string.
- `parse-integer`: Parses an integer from a string argument and returns this integer and the length of the string. Note, when the string is not an integer, this function will return an error. So, unless you're absolutely sure, use `read-from-string` instead.

```
(parse-integer "14")
=> 14
=> 2
```

- `make-keyword`: Returns a keyword of its arguments (strings, symbols and numbers). When the argument is a number, pipe-symbols "|" will be put around the number, before a keyword is created.

```
(make-keyword "a")
=> :a
(make-keyword 13)
=> :|13|
```

Regular Expressions

Regular Expressions are a set of functions which can be operated on strings. They can be used when reading in strings as input. The regular expression functions can be used to check and manipulate the strings for further handling. The functions are not desctructive. The three most used regular expression functions are listed below. For more information, refer to Franz Documentation.

- `match-re`: Regular Expression function, which detects symbols in a string (a certain character, but also whitespaces). Returns `t` and the detected symbol or `nil`. Extra optional keyword arguments are (also applicable for other two functions):
  - `:start` (integer), indicates the place the function should start in the string
  - `:end` (integer), indicates the place where the function should stop

```
(setq a "abcde")
(match-re "b" a)
=> t
=> "b"
(match-re "f" a)
=>nil
(match-re "a" a :start 3)
=>nil
```

- `replace-re`: Regular Expression function, which replaces symbols in a string by other symbols, given in the function arguments.

```
(setq a "abcde")
(replace-re a "b" "c")
=>"accde"
```

- `split-re`: Function to split a string into multiple strings at a certain symbol. This symbol is deleted, so not found back in one of the new strings. The function returns a list of the new strings.

```
(setq a "a#b#cde"}
(split-re "#" a)
=>("a" "b" "cde")
(split-re "#" a :start 3)
=>("a#b" "cde")
```

Next to stating the exact symbol (i.e. "#"), it is also possible to match, replace and split at a group of signals. The most important are:

- `\\d, \\d+` Detects any digit number, without the plus only once, with the plus multiple
- `\\s, \\s+` Detects white spaces, without the plus only once, with the plus multiple

Robustness

- `error`, `warn`: `error` and `warn` give top-level messages, while `error` interrupts evaluation and `warn` just continues. Use these functions already early on to make your program more robust and to ease debugging.
- `cerror`: throws a *correctable* error, consider it using it instead of `error`.
- `throw`, `catch`: When a symbol is `thrown` within the `catch`'s body, execution of the `catch`'s body is terminated and there is proceeded after `catch` s-expression.

```
(defun my-inverse (num)
  (catch 'error
    (when (zerop num) (throw 'error (pprint "Divisor is zero")))
    (pprint (/ 1 num)))
  (pprint "Function has ended"))
```

```
(my-inverse 3)
=> 1/3, "Function has ended"
(my-inverse 0)
=> "Divisor is zero", "Function has ended"
```

- `handler-case`: a function that catches expected conditions only, based on their type.
- `with-simple-restart`: provides a restart to the user.
- `unwind-protect`: evaluates a *protected form* and guarantees that *cleanup-forms* are executed before unwind-protect exits, whether it terminates normally or is aborted by a control transfer of some kind

I/O

Writing

- `print`, `pprint`, `pprinc`:
- `format`: `format` is a very, very powerful function for formatting strings to file, variables or command line. Most of it is explained in Graham, page 379-384.
  - Note that a single ~ allows you to ignore a new line in your code, such that you can keep within margins of 88 characters.
  - Note that you should not place string characters on the left side of your buffer. Use at least two whitespaces at the beginning of the line by default

- Also use capitals for format directives as it is more distinct: `~A` instead of `~a`.
- Learn useful idioms, e.g. `~{~A~^, ~}`, `~:p`. Learn when to use `~&` and `~%`. Also `~2%` and `~2&` are handy.

```
(format nil "A very, very, very ~
  long string."
=> "A very, very, very long string."
```

- `with-open-file`: Macro to write or read a specified stream to/from a file-pathname. For writing, the stream can be formatted with `format`. Important keywords are:
  - `:direction` To specify whether to read or write
  - `:if-exists` To specify the action when the file already exists
  - `:if-does-not-exists` To specify the action when the file does not exists

```
(with-open-file (stream file-pathname
                 :direction :output
                 :if-exists :supersede
                 :if-does-not-exists :create)
        (format stream .....))
```

The advantage over other functions (i.e. `open` and `close`) of this type, is the error-handling. After reading or writing, the stream is automatically closed, also when an error occurs in the evaluation of the body.

Reading

- `read`
- `read-from-string`: function to read expressions from a string. It stops when the first expression is read and returns this expression and the position in the string where it stopped reading.

```
(read-from-string "ab cde")
=> ab
=> 3
```

- `read-line`

System Operators

- `make-pathname`
- `merge-pathnames`
- `pathname-device`
- `pathname-directory`
- `pathname-name`
- `pathname-type`
- `namestring`
- `translate-logical-pathname`
- `make-directory`
- `rename-file`
- `delete-file`
- `delete-directory-and-files`
- `directory`: Same as `dir` in MS-DOS, returns a list of all files and folders in the directory.
- `probe-file`: Tests whether the file or folder that the pathname points to exists.

System

- `gc`: garbage collect
- excl.osi

    - copy-file
    - `getenv`: Function to retrieve Windows environment variables. E.g. `"temp"` or `"computername"`.

```
(excl.osi:getenv "computername")
=> "BECKETT"
```

    - `command-output`
- `decode-universal-time`: decodes the universal time into 9 values: second, minute, hour, date, month, year, day of the week, `t` or `nil` whether daylight saving time is in effect and the time-zone with respect to GMT.
- `get-universal-time`: returns the current time as number of seconds passed since 0:00:00 (GMT), January 1st, 1900.
- `iso-8601-date`: translates the universal time to date and (when requested) time

```
(iso-8601-date (get-universal-time) :include-time? t)
=>"2011-06-21T11:15:56"
```

Others

- `progn`: Function to evaluate each argument in order.
- `compose`, `disjoin`, `conjoin`, (`curry`, `rcurry`)
- `values`, `multiple-value-bind`, `multiple-value-list`: where `values` returns each argument in sequence, `multiple-value-bind` lets you use them in a local environment and `multiple-value-list` immediately puts them into a list. These functions are easily combined with the `floor`, `truncate`, `ceiling` and `round` functions.

```
(values 3 2)
=> 3
=> 2
```

```
(multiple-value-bind (q r)
                     (floor 11 3)
                   (expt q r))
=> 9
```

```
(multiple-value-list (floor 11 3))
=> (3 2)
```

- `eval`
- `getf`
- `maphash`

- `sys:resize-areas`
- `exit`
- `gwl:clear-instance`
- `gwl:define-package`

GDL functions

Object definition

- `define-object`

Object creation

- `make-object`, `make-self`

Pointing

- `set-self`
- `the`, `the-object`, `the-child`, `the-element`

Bashing

- `set-slot!`, `set-slots!`

Geometry

- `make-point`
- `add-vectors`
- `translate`, `translate-along-vector`

I/O

- `with-cl-who-string`: Macro to turn LISP into HTML or XML string. Now mostly used to define main-views of objects in the User Interface. For more information, refer to CL-WHO.