```
* (get-universal-time)

* (decode-universal-time 3220993326)

* (get-decoded-time)

* (decode-universal-time (get-universal-time))


* (defconstant *day-names*
    '("Monday" "Tuesday" "Wednesday"
      "Thursday" "Friday" "Saturday"
      "Sunday"))


* (multiple-value-bind
        (second minute hour date month year day-of-week dst-p tz)
        (get-decoded-time)
    (format t "It is now ~2,'0d:~2,'0d:~2,'0d of ~a, ~d/~2,'0d/~d (GMT~@d)"
             hour
             minute
             second
             (nth day-of-week *day-names*)
             month
             date
             year
             (- tz)))


* (encode-universal-time 6 22 19 25 1 2002)

* (setq *moon* (encode-universal-time 0 17 16 20 7 1969 4))

* (setq *takeoff* (encode-universal-time 0 38 11 28 1 1986 5))

* (- *takeoff* *moon*)

* internal-time-units-per-second

* (let ((real1 (get-internal-real-time))
        (run1 (get-internal-run-time)))
    (... your call here ...)
    (let ((run2 (get-internal-run-time))
          (real2 (get-internal-real-time)))
        (format t "Computation took:~%")
        (format t "  ~f seconds of real time~%"
                (/ (- real2 real1) internal-time-units-per-second))
        (format t "  ~f seconds of run time~%"
                (/ (- run2 run1) internal-time-units-per-second))))



* (defmacro timing (&body forms)
    (let ((real1 (gensym))
          (real2 (gensym))
          (run1 (gensym))
          (run2 (gensym))
          (result (gensym)))
    `(let* ((,real1 (get-internal-real-time))
             (,run1 (get-internal-run-time))
             (,result (progn ,@forms))
```

```
              (,run2 (get-internal-run-time))
              (,real2 (get-internal-real-time)))
         (format *debug-io* ";;; Computation took:~%")
         (format *debug-io* ";;;  ~f seconds of real time~%"
                 (/ (- ,real2 ,real1) internal-time-units-per-second))
         (format t ";;;  ~f seconds of run time~%"
                 (/ (- ,run2 ,run1) internal-time-units-per-second))
         ,result)))


* (timing (sleep 1))


* (let ((numbers (loop for i from 1 to 100 collect (random 1.0))))
    (time (sort numbers #'<)))




* (defun day-of-week (day month year)
    "Returns the day of the week as an integer.
Monday is 0."
    (nth-value
     6
     (decode-universal-time
      (encode-universal-time 0 0 0 day month year 0)
      0)))


* (day-of-week 23 12 1965)


* (day-of-week 1 1 1900)


* (day-of-week 31 12 1899)



(defun day-of-week (day month year)
  "Returns the day of the week as an integer.
Sunday is 0. Works for years after 1752."
  (let ((offset '(0 3 2 5 0 3 5 1 4 6 2 4)))
    (when (< month 3)
      (decf year 1))
    (mod
     (truncate (+ year
                  (/ year 4)
                  (/ (- year)
                     100)
                  (/ year 400)
                  (nth (1- month) offset)
                  day
                  -1))
     7)))
```

```
* (defparameter *my-hash* (make-hash-table))


* (setf (gethash 'one-entry *my-hash*) "one")


* (setf (gethash 'another-entry *my-hash*) 2/4)


* (gethash 'one-entry *my-hash*)

* (gethash 'another-entry *my-hash*)


* (defparameter *my-hash* (make-hash-table))

* (setf (gethash 'one-entry *my-hash*) "one")

* (if (gethash 'one-entry *my-hash*)
    "Key exists"
    "Key does not exist")

* (if (gethash 'another-entry *my-hash*)
    "Key exists"
    "Key does not exist")


* (setf (gethash 'another-entry *my-hash*) nil)


* (if (gethash 'another-entry *my-hash*)
    "Key exists"
    "Key does not exist")


* (if (nth-value 1 (gethash 'another-entry *my-hash*))
    "Key exists"
    "Key does not exist")


* (if (nth-value 1 (gethash 'no-entry *my-hash*))
    "Key exists"
    "Key does not exist")


* (defparameter *my-hash* (make-hash-table))

* (setf (gethash 'first-key *my-hash*) 'one)

* (gethash 'first-key *my-hash*)

* (remhash 'first-key *my-hash*)

* (gethash 'first-key *my-hash*)

* (gethash 'no-entry *my-hash*)
* (remhash 'no-entry *my-hash*)

* (gethash 'no-entry *my-hash*)


* (defparameter *my-hash* (make-hash-table))

* (setf (gethash 'first-key *my-hash*) 'one)

* (setf (gethash 'second-key *my-hash*) 'two)

* (setf (gethash 'third-key *my-hash*) nil)

* (setf (gethash nil *my-hash*) 'nil-value)

* (defun print-hash-entry (key value)
    (format t "The value associated with the key ~S is ~S~%" key value))


* (maphash #'print-hash-entry *my-hash*)

* (with-hash-table-iterator (my-iterator *my-hash*)
    (loop
      (multiple-value-bind (entry-p key value)
          (my-iterator)
        (if entry-p
            (print-hash-entry key value)
            (return)))))



* (loop for key being the hash-keys of *my-hash*
        do (print key))



* (loop for key being the hash-keys of *my-hash*
        using (hash-value value)
        do (format t "The value associated with the key ~S is ~S~%" key
value))

* (loop for value being the hash-values of *my-hash*
        do (print value))

* (loop for value being the hash-values of *my-hash*
        using (hash-key key)
        do (format t "~&~A -> ~A" key value))

* (defparameter *my-hash* (make-hash-table))

* (hash-table-count *my-hash*)

* (setf (gethash 'first *my-hash*) 1)

* (setf (gethash 'second *my-hash*) 2)

* (setf (gethash 'third *my-hash*) 3)

* (hash-table-count *my-hash*)
* (setf (gethash 'second *my-hash*) 'two)


* (hash-table-count *my-hash*)


* (clrhash *my-hash*)


* (hash-table-count *my-hash*)


* (defparameter *my-hash* (make-hash-table))


* (hash-table-size *my-hash*)


* (hash-table-rehash-size *my-hash*)


* (time (dotimes (n 100000) (setf (gethash n *my-hash*) n)))



* (let ((size 65)) (dotimes (n 20) (print (list n size)) (setq size (* 1.5
size))))

* (defparameter *my-hash* (make-hash-table :size 100000))

* (hash-table-size *my-hash*)

* (time (dotimes (n 100000) (setf (gethash n *my-hash*) n)))

* (defparameter *my-hash* (make-hash-table :rehash-size 100000))

* (hash-table-size *my-hash*)

* (hash-table-rehash-size *my-hash*)

* (time (dotimes (n 100000) (setf (gethash n *my-hash*) n)))
* (loop for x in '(a b c d e)
      do (print x) )



* (loop for x in '(a b c d e)
      for y in '(1 2 3 4 5)
      collect (list x y) )


* (loop for x from 1 to 5
      for y = (* x 2)
      collect y)


* (loop for x in '(a b c d e)
      for y from 1

      when (> y 1)
      do (format t ", ")

      do (format t "~A" x)
      )




* (loop for x in '(a b c d e)
      for y from 1

      if (> y 1)
      do (format t ", ~A" x)
      else do (format t "~A" x)
      )


* (loop for x in '(a b c d e 1 2 3 4)
        until (numberp x)
        collect (list x 'foo))




* (loop for x from 1
      for y = (* x 10)
      while (< y 100)

      do (print (* x 5))

      collect y)
```

```
* (loop for x from 1 to 10
      collect (loop for y from 1 to x
                   collect y) )
```

```
* (loop for (a b) in '((x 1) (y 2) (z 3))
      collect (list b a) )
```

```
* (let ((s "alpha45"))
  (loop for i from 0 below (length s)
       for ch =  (char s i)
       when (find ch "0123456789" :test #'eql)
       return ch) )
```

```
* (loop for x in '(foo 2)
      thereis (numberp x))
```

```
* (loop for x in '(foo 2)
      never (numberp x))
```

```
* (loop for x in '(foo 2)
      always (numberp x))
```

```
* (loop

  for element in (list 1 2 3 4 5 6) do wyrażenie ...)
```

```
* (loop

  for c across string collect c)
```

```
* (loop

  for i in list1 and j in list2 collect (list i j))
```

```
* (loop

  for (k . v) in (pairlis '(a b c) '(1 2 3)) do
  (format t "~a => ~a~%" k v))

* (loop
  for para on (list 1 2 3 4 5 6) do
  (format t "~a => ~a~%" (car para) (cadr para)))

* (setq x 10)
(let ((x 20))
  (* x 30))


* (let* ((x 10)
       (y (+ x 10)))
    (* x y))


* (defconstant +pi+ 3.14159265358979)

* (dotimes (i 10)
  (print i))

* (do ((i 0 (incf i))
      (j 10 (decf j)))
    ((zerop j) 'done)
    (print (+ i j)))

* (let ((i 10))
  (loop
   (when (zerop i) (return))
   (print (decf i))))
```

```
* (loop
  for i in (list 0 1 2 3 4 5 6)
  when (evenp i) collect i)

* (loop
  for i from 0 while (< i 10) collect i)


* (setq x (quote foo))

* (setq y 'foo)

* "jakiś napis"

* (make-string 10)

* (parse-integer "256")

* (char-code #\a)

* (code-char 100)


* (setq array (make-array '(4 4) :initial-element 0))

* (setf (aref array 1 1) 1)

* (setq aa (make-array 10 :adjustable t :fill-pointer 0))
```