Programming Language—Common Lisp

6. Iteration

6.1 The LOOP Iteration Facility

The **loop** *macro* performs iteration.

loop *forms* are partitioned into two categories: simple **loop** *forms* and extended **loop** *forms*.

6.1.1 Simple Loop

A simple **loop** *form* is one that has a body containing only *compound forms*. Each *form* is *evaluated* in turn from left to right. When the last *form* has been *evaluated*, then the first *form* is evaluated again, and so on, in a never-ending cycle. A simple **loop** *form* establishes an *implicit block* named **nil**. The execution of a simple **loop** can be terminated by explicitly transfering control to the *implicit block* (using **return** or **return-from**) or to some *exit point* outside of the *block* (*e.g.*, using **throw**, **go**, or **return-from**).

6.1.2 Extended Loop

An extended **loop** *form* is one that has a body containing *atomic expressions*. When the **loop** *macro* processes such a *form*, it invokes a facility that is commonly called "the Loop Facility."

The Loop Facility provides standardized access to mechanisms commonly used in iterations through Loop schemas, which are introduced by *loop keywords*.

The body of an extended **loop** *form* is divided into **loop** clauses, each which is in turn made up of *loop keywords* and *forms*.

6.1.2.1 Loop Keywords

Loop keywords are not true *keywords*$_1$; they are special *symbols*, recognized by *name* rather than *object* identity, that are meaningful only to the **loop** facility. A *loop keyword* is a *symbol* but is recognized by its *name* (not its identity), regardless of the *packages* in which it is *accessible*.

In general, *loop keywords* are not *external symbols* of the `COMMON-LISP` *package*, except in the coincidental situation that a *symbol* with the same name as a *loop keyword* was needed for some other purpose in Common Lisp. For example, there is a *symbol* in the `COMMON-LISP` *package* whose *name* is `"UNLESS"` but not one whose *name* is `"UNTIL"`.

If no *loop keywords* are supplied in a **loop** *form*, the Loop Facility executes the loop body repeatedly; see Section 6.1.1 (Simple Loop).

6.1.2.2 Overview of the Loop Facility

A **loop** *macro form* expands into a *form* containing one or more binding forms (that *establish* *bindings* of loop variables) and a **block** and a **tagbody** (that express a looping control structure). The variables established in **loop** are bound as if by **let** or **lambda**.

Implementations can interleave the setting of initial values with the *bindings*. However, the assignment of the initial values is always calculated in the order specified by the user. A variable is thus sometimes bound to a meaningless value of the correct *type*, and then later in the prologue it is set to the true initial value by using **setq**. One implication of this interleaving is that it is *implementation-dependent* whether the *lexical environment* in which the initial value *forms* (variously called the *form1*, *form2*, *form3*, *step-fun*, *vector*, *hash-table*, and *package*) in any *for-as-subclause*, except *for-as-equals-then*, are *evaluated* includes only the loop variables preceding that *form* or includes more or all of the loop variables; the *form1* and *form2* in a *for-as-equals-then* form includes the *lexical environment* of all the loop variables.

After the *form* is expanded, it consists of three basic parts in the **tagbody**: the loop prologue, the loop body, and the loop epilogue.

Loop prologue

The loop prologue contains *forms* that are executed before iteration begins, such as any automatic variable initializations prescribed by the *variable* clauses, along with any `initially` clauses in the order they appear in the source.

Loop body

The loop body contains those *forms* that are executed during iteration, including application-specific calculations, termination tests, and variable *stepping*$_1$.

Loop epilogue

The loop epilogue contains *forms* that are executed after iteration terminates, such as `finally` clauses, if any, along with any implicit return value from an *accumulation* clause or an *end-test* clause.

Some clauses from the source *form* contribute code only to the loop prologue; these clauses must come before other clauses that are in the main body of the **loop** form. Others contribute code only to the loop epilogue. All other clauses contribute to the final translated *form* in the same order given in the original source *form* of the **loop**.

Expansion of the **loop** macro produces an *implicit block* named **nil** unless `named` is supplied. Thus, **return-from** (and sometimes **return**) can be used to return values from **loop** or to exit **loop**.

6.1.2.3 Syntax of an Extended LOOP Form

loop [`named` *name*] [`variables`]* [`main`]*

variables::= `with` | *initial-final* | `for` | `as` *initial-final*::= `initially` | `finally`

main::= *unconditional* | *accumulation* | *conditional* | *end-test* | *initial-final*

unconditional::= `do` | `doing` | `return`

accumulation::= `collect` | `collecting` | `append` | `appending`
| `nconc` | `nconcing` | `count` | `counting` | `sum` | `summing`
| `maximize` | `maximizing` | `minimize` | `minimizing`

conditional::= `when` | `if` | `unless`

end-test::= `while` | `until` | `always` | `never` | `thereis` | `repeat`

6.1.2.3.1 Syntax for loop clauses

The syntax for loop clauses is represented in the following way:

- The *loop keyword* name is in code font.

- A word in argument font indicates a syntactic type that is included in the loop clause definition, the *type* of which is given in the Argument section (for example, *symbol*, *argument*, *variable*).

`named` *name*

`with` *var1* [*type-spec*] [= *form1*]
{and *var2* [*type-spec*] [= *form2*]}*

- *for-as-clause*::= *for-as-subclause* {and *for-as-subclause*}*

- *for-as-subclause*::= {*for-as-arithmetic* | *for-as-in-list* | *for-as-on-list* | *for-as-equals-then* | *for-as-across* | *for-as-hash* | *for-as-package*}

- *for-as-arithmetic*::= {`for` | `as`} *var* [*type-spec*] [{`from` | `downfrom` | `upfrom`} *form1*]
[{`to` | `downto` | `upto` | `below` | `above`} *form2*]
[by *form3*]

- *for-as-in-list*::= {`for` | `as`} *var* [*type-spec*] `in` *form1* [by *step-fun*]

- *for-as-on-list*::= {`for` | `as`} *var* [*type-spec*] `on` *form1* [by *step-fun*]

- *for-as-equals-then*::= {`for` | `as`} *var* [*type-spec*] = *form1* [then *form2*]

- *for-as-across*::= {`for` | `as`} *var* [*type-spec*] `across` *vector*

- *for-as-hash*::= {for | as} *var* [*type-spec*] being {each | the}
  {hash-key[s] | hash-value[s]}
  {in | of} *hash-table*
  [using ({hash-key | hash-value} *other-var*)]
- *for-as-package*::= {for | as} *var* [*type-spec*] being {each | the}
  {symbol[s] | present-symbol[s] | external-symbol[s]}
  [{in | of} *package*]

initially [*form*]+

finally [*form*]+

finally *unconditional-clause*

{do | doing} [*form*]+

return *form*

{collect | collecting} *form* [into *var*]

{append | appending} *form* [into *var*]

{nconc | nconcing} *form* [into *var*]

{count | counting} *form* [into *var*] [*type-spec*]

{sum | summing} *form* [into *var*] [*type-spec*]

{maximize | maximizing} *form* [into *var*] [*type-spec*]

{minimize | minimizing} *form* [into *var*] [*type-spec*]

{if | when | unless} *form clause1* [and *clause*]*
[end]

{if | when | unless} *form clause1* [and *clause*]*
else *clause2* [and *clause*]*
[end]

while *form*

until *form*

always *form*

never *form*

thereis *form*

repeat *form*

6.1.2.3.1.1 Argument Restrictions

name—a *symbol*.

var, *var1*, *var2*, *other-var*—a *symbol* (a *variable name*).

form, *form1*, *form2*, *form3*—a *form*.

step-fun—a *designator* for a *function* of one *argument*.

vector—a *vector*.

hash-table—a *hash table*.

package—a *package designator*.

unconditional-clause—a `do`, `doing`, or `return` clause.

Clause, *clause1* – `it` | *loop keyword* followed by *forms* | *loop keyword* followed by *forms* in addition to possibly being followed by other *loop keywords* and *forms*.

type-spec::= `of-type` *d-type-spec* | `fixnum` | `float` | `t` | `nil`

d-type-spec::= *type-specifier* | (*d-type-spec* . *d-type-spec*)

type-specifier—a *type specifier*.

6.1.2.4 Parsing Loop Clauses

The syntactic parts of a **loop** *form* are called clauses; the rules for parsing are determined by that clause's keyword. The following example shows a **loop** *form* with six clauses:

```
(loop for i from 1 to (compute-top-value)       ; first clause
      while (not (unacceptable i))              ; second clause
      collect (square i)                        ; third clause
      do (format t "Working on ~D now" i)       ; fourth clause
      when (evenp i)                            ; fifth clause
        do (format t "~D is a non-odd number" i)
      finally (format t "About to exit!"))      ; sixth clause
```

Each *loop keyword* introduces either a compound loop clause or a simple loop clause that can consist of a *loop keyword* followed by a single *form*. The number of *forms* in a clause is determined by the *loop keyword* that begins the clause and by the auxiliary keywords in the clause. The keywords `do`, `initially`, and `finally` are the only loop keywords that can take any number of *forms* and group them as an *implicit progn*.

Loop clauses can contain auxiliary keywords, which are sometimes called prepositions. For example, the first clause in the code above includes the prepositions `from` and `to`, which mark the value from which stepping begins and the value at which stepping ends.

6.1.2.5 Kinds of Loop Clauses

Loop clauses fall into one of the following categories:

6.1.2.5.1 Variable Initialization and Stepping Clauses

- The `for` and `as` constructs provide iteration control clauses that establish a variable to be initialized. `for` and `as` clauses can be combined with the loop keyword `and` to get *parallel* initialization and *stepping*$_1$. Otherwise, the initialization and *stepping*$_1$ are *sequential*.

- The `with` construct is similar to a single `let` clause. `with` clauses can be combined using the *loop keyword* `and` to get *parallel* initialization.

6.1.2.5.2 Value Accumulation Clauses

- The `collect` construct takes one *form* in its clause and adds the value of that *form* to the end of a *list* of values. By default, the *list* of values is returned when the **loop** finishes.

- The `append` construct takes one *form* in its clause and appends the value of that *form* to the end of a *list* of values. By default, the *list* of values is returned when the **loop** finishes.

- The `nconc` construct is similar to the `append` construct, but its *list* values are concatenated as if by the function `nconc`. By default, the *list* of values is returned when the **loop** finishes.

- The `sum` construct takes one *form* in its clause that must evaluate to a *number* and accumulates the sum of all these *numbers*. By default, the cumulative sum is returned when the **loop** finishes.

- The `count` construct takes one *form* in its clause and counts the number of times that the *form* evaluates to *true*. By default, the count is returned when the **loop** finishes.

- The `minimize` construct takes one *form* in its clause and determines the minimum value obtained by evaluating that *form*. By default, the minimum value is returned when the **loop** finishes.

- The `maximize` construct takes one *form* in its clause and determines the maximum value obtained by evaluating that *form*. By default, the maximum value is returned when the **loop** finishes.

6.1.2.5.3 Termination Condition Clauses

– The `for` and `as` constructs provide a termination test that is determined by the iteration control clause.

– The `repeat` construct causes termination after a specified number of iterations. (It uses an internal variable to keep track of the number of iterations.)

– The `while` construct takes one *form*, a *condition*, and terminates the iteration if the *condition* evaluates to *false*. A `while` clause is equivalent to the expression `(if (not` *condition*`) (loop-finish))`.

– The `until` construct is the inverse of `while`; it terminates the iteration if the *condition* evaluates to any *non-nil* value. An `until` clause is equivalent to the expression `(if` *condition* `(loop-finish))`.

– The `always` construct takes one *form* and terminates the **loop** if the *form* ever evaluates to *false*; in this case, the **loop** *form* returns **nil**. Otherwise, it provides a default return value of `t`.

– The `never` construct takes one *form* and terminates the **loop** if the *form* ever evaluates to *true*; in this case, the **loop** *form* returns **nil**. Otherwise, it provides a default return value of `t`.

– The `thereis` construct takes one *form* and terminates the **loop** if the *form* ever evaluates to a *non-nil object*; in this case, the **loop** *form* returns that *object*.

If multiple terminate condition clauses are specified, the **loop** *form* terminates if any are satisfied.

Note also that the **loop-finish** macro terminates iteration and returns any accumulated result. Any `finally` clauses that are supplied are evaluated.

6.1.2.5.4 Unconditional Execution Clauses

– The `do` construct evaluates all *forms* in its clause.

– The `return` construct takes one *form* and returns its value. It is equivalent to the clause `do (return` *value*`)`.

6.1.2.5.5 Conditional Execution Clauses

– The `if` and `when` constructs take one *form* as a predicate and a clause that is executed when the predicate is *true*. The clause can be a value accumulation, unconditional, or another conditional clause; it can also be any combination of such clauses connected by the **loop** `and` keyword.

– The **loop** `unless` construct is similar to the **loop** `when` construct except that it complements the predicate.

– The **loop** `else` construct provides an optional component of `if`, `when`, and `unless` clauses that is executed when an `if` or `when` predicate *yields false* or when an `unless` predicate *yields true*. The component is one of the clauses described under `if`.

– The **loop** `end` construct provides an optional component to mark the end of a conditional clause.

6.1.2.5.6 Miscellaneous Clauses

– The **loop** `named` construct gives a name for the *block* of the loop.

– The **loop** `initially` construct causes its *forms* to be evaluated in the loop prologue, which precedes all **loop** code except for initial settings supplied by the constructs `with`, `for`, or `as`.

– The **loop** `finally` construct causes its *forms* to be evaluated in the loop epilogue after normal iteration terminates. An unconditional clause can also follow the **loop** `finally` keyword.

6.1.2.6 Loop Facility

The Loop Facility includes the following operations.

Iteration Control

Iteration control clauses allow direction of **loop** iteration. The *loop keywords* `for` and `as` designate iteration control clauses. Iteration control clauses differ with respect to the specification of termination conditions and to the initialization and *stepping*$_1$ of loop variables. Iteration clauses by themselves do not cause the Loop Facility to return values, but they can be used in conjunction with value-accumulation clauses to return values.

All variables are initialized in the loop prologue. A *variable binding* has *lexical scope* unless it is proclaimed **special**; thus, by default, the variable can be *accessed* only by *forms* that lie textually within the **loop**. Stepping assignments are made in the loop body before any other *forms* are evaluated in the body.

The variable argument in iteration control clauses can be a destructuring list. A destructuring list is a *tree* whose *non-nil atoms* are *variable names*. See Section 6.1.2.8 (Destructuring).

The iteration control clauses `for`, `as`, and `repeat` must precede any other loop clauses, except `initially`, `with`, and `named`, since they establish variable *bindings*. When iteration control clauses are used in a **loop**, the corresponding termination tests in the loop body are evaluated before any other loop body code is executed.

If multiple iteration clauses are used to control iteration, variable initialization and *stepping*$_1$ occur *sequentially* by default. The `and` construct can be used to connect two or more iteration clauses when *sequential binding* and *stepping*$_1$ are not necessary. The iteration behavior of clauses joined by `and` is analogous to the behavior of the macro **do** with respect to **do***.

The `for` and `as` clauses iterate by using one or more local loop variables that are initialized to some value and that can be modified or *stepped*$_1$ after each iteration. For these clauses, iteration terminates when a local variable reaches some supplied value or when some other loop clause terminates iteration. At each iteration, variables can be *stepped*$_1$ by an increment or a decrement or can be assigned a new value by the evaluation of a *form*). Destructuring can be used to assign values to variables during iteration.

The `for` and `as` keywords are synonyms; they can be used interchangeably. There are seven syntactic formats for these constructs. In each syntactic format, the *type* of *var* can be supplied by the optional *type-spec* argument. If *var* is a destructuring list, the *type* supplied by the *type-spec* argument must appropriately match the elements of the list. By convention, `for` introduces new iterations and `as` introduces iterations that depend on a previous iteration specification.

for-as-arithmetic

In the *for-as-arithmetic* subclause, the `for` or `as` construct iterates from the value supplied by *form1* to the value supplied by *form2* in increments or decrements denoted by *form3*. Each expression is evaluated only once and must evaluate to a *number*. The variable *var* is bound to the value of *form1* in the first iteration and is *stepped*$_1$ by the value of *form3* in each succeeding iteration, or by 1 if *form3* is not provided. The following *loop keywords* serve as valid prepositions within this syntax. At least one of the prepositions must be used; and at most one from each line may be used in a single subclause.

```
from | downfrom | upfrom

to | downto | upto | below | above

by
```

The descriptions of the prepositions follow:

`from`

The *loop keyword* `from` specifies the value from which *stepping*$_1$ begins, as supplied by *form1*. *Stepping*$_1$ is incremental by default. If decremental *stepping*$_1$ is desired, the preposition `downto` or `above` must be used with *form2*. For incremental *stepping*$_1$, the default `from` value is 0.

`downfrom, upfrom`

The *loop keyword* `downfrom` indicates that the variable *var* is decreased in decrements supplied by *form3*; the *loop keyword* `upfrom` indicates that *var* is increased in increments supplied by *form3*.

`to`

The *loop keyword* `to` marks the end value for *stepping*$_1$ supplied in *form2*. *Stepping*$_1$ is incremental by default. If decremental *stepping*$_1$ is desired, the preposition `downfrom` must be used with *form1*, or else the preposition `downto` or `above` should be used instead of `to` with *form2*.

`downto, upto`

The *loop keyword* `downto` specifies decremental *stepping*; the *loop keyword* `upto` specifies incremental *stepping*. In both cases, the amount of change on each step is specified by *form3*, and the **loop** terminates when the variable *var* passes the value of *form2*. Since there is no default for *form1* in decremental *stepping*$_1$, a *form1* value must be supplied (using `from` or `downfrom`) when `downto` is supplied.

`below, above`

The *loop keywords* `below` and `above` are analogous to `upto` and `downto` respectively. These keywords stop iteration just before the value of the variable *var* reaches the value supplied by *form2*; the end value of *form2* is not included. Since there is no default for *form1* in decremental *stepping*$_1$, a value must be supplied with `above`.

`by`

The *loop keyword* `by` marks the increment or decrement supplied by *form3*. The value of *form3* can be any positive *number*. The default value is 1.

In an iteration control clause, the `for` or `as` construct causes termination when the supplied limit is reached. That is, iteration continues until the value *var* is stepped to the exclusive or inclusive limit supplied by *form2*. The range is exclusive if *form3* increases or decreases *var* to the value of *form2* without reaching that value; the loop keywords `below` and `above` provide exclusive limits. An inclusive limit allows *var* to attain the value of *form2*; `to`, `downto`, and `upto` provide inclusive limits.

for-as-in-list

In the *for-as-in-list* subclause, the `for` or `as` construct iterates over the contents of a *list*. It checks for the end of the *list* as if by using **endp**. The variable *var* is bound to the successive elements of the *list* in *form1* before each iteration. At the end of each iteration, the function *step-fun* is applied to the *list*; the default value for *step-fun* is **cdr**. The *loop keywords* `in` and `by` serve as valid prepositions in this syntax. The `for` or `as` construct causes termination when the end of the *list* is reached.

for-as-on-list

In the *for-as-on-list* subclause, the `for` or `as` construct iterates over a *list*. It checks for the end of the *list* as if by using **atom**. The variable *var* is bound to the successive tails of the *list* in *form1*. At the end of each iteration, the function *step-fun* is applied to the *list*; the default value for *step-fun* is **cdr**. The *loop keywords* `on` and `by` serve as valid prepositions in this syntax. The `for` or `as` construct causes termination when the end of the *list* is reached.

for-as-equals-then

In the *for-as-equals-then* subclause the `for` or `as` construct initializes the variable *var* by setting it to the result of evaluating *form1* on the first iteration, then setting it to the result of evaluating *form2* on the second and subsequent iterations. If *form2* is omitted, the construct uses *form1* on the second and subsequent iterations. The *loop keywords* = and `then` serve as valid prepositions in this syntax. This construct does not provide any termination conditions.

for-as-across

In the *for-as-across* subclause the `for` or `as` construct binds the variable *var* to the value of each element in the array *vector*. The *loop keyword* `across` marks the array *vector*; `across` is used as a preposition in this syntax. Iteration stops when there are no more elements in the supplied *array* that can be referenced. Some implementations might recognize a **the** special form in the *vector* form to produce more efficient code.

for-as-hash

In the *for-as-hash* subclause the `for` or `as` construct iterates over the elements, keys, and values of a *hash-table*. In this syntax, a compound preposition is used to designate access to a *hash table*. The variable *var* takes on the value of each hash key or hash value in the supplied *hash-table*. The following *loop keywords* serve as valid prepositions within this syntax:

`being`

The keyword `being` introduces either the Loop schema `hash-key` or `hash-value`.

`each`, `the`

The *loop keyword* `each` follows the *loop keyword* `being` when `hash-key` or `hash-value` is used. The *loop keyword* `the` is used with `hash-keys` and `hash-values` only for ease of reading. This agreement isn't required.

`hash-key`, `hash-keys`

These *loop keywords* access each key entry of the *hash table*. If the name `hash-value` is supplied in a `using` construct with one of these Loop schemas, the iteration can optionally access the keyed value. The order in which the keys are accessed is undefined; empty slots in the *hash table* are ignored.

`hash-value`, `hash-values`

These *loop keywords* access each value entry of a *hash table*. If the name `hash-key` is supplied in a `using` construct with one of these Loop schemas, the iteration can optionally access the key that corresponds to the value. The order in which the keys are accessed is undefined; empty slots in the *hash table* are ignored.

`using`

The *loop keyword* `using` introduces the optional key or the keyed value to be accessed. It allows access to the hash key if iteration is over the hash values, and the hash value if iteration is over the hash keys.

`in`, `of`

These loop prepositions introduce *hash-table*.

In effect

`being` [`each|the`] [`hash-value|hash-values|hash-key|hash-key`] [`in|of`]

is a compound preposition.

Iteration stops when there are no more hash keys or hash values to be referenced in the supplied *hash-table*.

for-as-package

In the *for-as-package* subclause the `for` or `as` construct iterates over the *symbols* in a *package*. In this syntax, a compound preposition is used to designate access to a *package*. The variable *var* takes on the value of each *symbol* in the supplied *package*. The following *loop keywords* serve as valid prepositions within this syntax:

`being`

The keyword `being` introduces either the Loop schema `symbol`, `present-symbol`, or `external-symbol`.

`each`, `the`

The *loop keyword* `each` follows the *loop keyword* `being` when `symbol`, `present-symbol`, or `external-symbol` is used. The *loop keyword* `the` is used with `symbols`, `present-symbols`, and `external-symbols` only for ease of reading. This agreement isn't required.

`present-symbol`, `present-symbols`

These Loop schemas iterate over the *symbols* that are *present* in a *package*. The *package* to be iterated over is supplied in the same way that *package* arguments to **find-package** are supplied. If the *package* for the iteration is not supplied, the *current package* is used. If a *package* that does not exist is supplied, an error of *type* **package-error** is signaled.

`symbol`, `symbols`

These Loop schemas iterate over *symbols* that are *accessible* in a given *package*. The *package* to be iterated over is supplied in the same way that *package* arguments to **find-package** are supplied. If the *package* for the iteration is not supplied, the *current package* is used. If a *package* that does not exist is supplied, an error of *type* **package-error** is signaled.

`external-symbol`, `external-symbols`

These Loop schemas iterate over the *external symbols* of a *package*. The *package* to be iterated over is supplied in the same way that *package* arguments to **find-package** are supplied. If the *package* for the iteration is not supplied, the *current package* is used. If a *package* that does not exist is supplied, an error of *type* **package-error** is signaled.

`in`, `of`

These loop prepositions introduce *package*.

In effect

`being` [`each`|`the`] [[[`present`|`external`]`symbol`] | [[`present`|`external`] `symbols`]] [`in`|`of`]

is a compound preposition.

Iteration stops when there are no more *symbols* to be referenced in the supplied *package*.

End-Test Control

The `repeat` construct causes iteration to terminate after a specified number of times. The loop body executes n times, where n is the value of the expression *form*. The *form* argument is evaluated one time in the loop prologue. If the expression evaluates to 0 or to a negative *number*, the loop body is not evaluated.

The constructs `always`, `never`, `thereis`, `while`, `until`, and the macro **loop-finish** allow conditional termination of iteration within a **loop**.

The constructs `always`, `never`, and `thereis` provide specific values to be returned when a **loop** terminates. Using `always`, `never`, or `thereis` in a loop with value accumulation clauses that are not `into` causes an error of *type* **program-error** to be signaled (at macro expansion time). Since `always`, `never`, and `thereis` use the **return-from** *special operator* to terminate iteration, any `finally` clause that is supplied is not evaluated when exit occurs due to any of these constructs. In all other respects these constructs behave like the `while` and `until` constructs.

The `always` construct takes one *form* and terminates the **loop** if the *form* ever evaluates to **nil**; in this case, it returns **nil**. Otherwise, it provides a default return value of `t`. If the value of the supplied *form* is never **nil**, some other construct can terminate the iteration.

The `never` construct terminates iteration the first time that the value of the supplied *form* is *non-nil*; the **loop** returns **nil**. If the value of the supplied *form* is always **nil**, some other construct can terminate the iteration. Unless some other clause contributes a return value, the default value returned is `t`.

The `thereis` construct terminates iteration the first time that the value of the supplied *form* is *non-nil*; the **loop** returns the value of the supplied *form*. If the value of the supplied *form* is always **nil**, some other construct can terminate the iteration. Unless some other clause contributes a return value, the default value returned is **nil**.

There are two differences between the `thereis` and `until` constructs:

- The `until` construct does not return a value or **nil** based on the value of the supplied *form*.

- The `until` construct executes any `finally` clause. Since `thereis` uses the **return-from** *special operator* to terminate iteration, any `finally` clause that is supplied is not evaluated when exit occurs due to `thereis`.

The `while` construct allows iteration to continue until the supplied *form* evaluates to *false*. The supplied *form* is reevaluated at the location of the `while` clause.

The `until` construct is equivalent to `while (not` *form*`)`.... If the value of the supplied *form* is *non-nil*, iteration terminates.

The `while` and `until` constructs can be used at any point in a **loop**. If an `until` or `while` clause causes termination, any clauses that precede it in the source are still evaluated. If the `until` and `while` constructs cause termination, control is passed to the loop epilogue, where any `finally` clauses will be executed.

There are two differences between the `never` and `until` constructs:

- The `until` construct does not return `t` or **nil** based on the value of the supplied *form*.

- The `until` construct does not bypass any `finally` clauses. Since `never` uses the

return-from *special operator* to terminate iteration, any `finally` clause that is supplied is not evaluated when exit occurs due to `never`.

In most cases it is not necessary to use **loop-finish** because other loop control clauses terminate the **loop**. The macro **loop-finish** is used to provide a normal exit from a nested condition inside a **loop**. In normal termination, `finally` clauses are executed and default return values are returned. Since **loop-finish** transfers control to the loop epilogue, using **loop-finish** within a `finally` expression can cause infinite looping.

End-test control constructs can be used anywhere within the loop body. The termination conditions are tested in the order in which they appear.

Value Accumulation

The constructs `collect`, `collecting`, `append`, `appending`, `nconc`, `nconcing`, `count`, `counting`, `maximize`, `maximizing`, `minimize`, `minimizing`, `sum`, and `summing`, allow values to be accumulated in a **loop**.

The constructs `collect`, `collecting`, `append`, `appending`, `nconc`, and `nconcing`, designate clauses that accumulate values in *lists* and return them. The constructs `count`, `counting`, `maximize`, `maximizing`, `minimize`, `minimizing`, `sum`, and `summing` designate clauses that accumulate and return numerical values.

During each iteration, the constructs `collect` and `collecting` collect the value of the supplied *form* into a *list*. When iteration terminates, the *list* is returned. The argument *var* is set to the *list* of collected values; if *var* is supplied, the **loop** does not return the final *list* automatically. If *var* is not supplied, it is equivalent to supplying an internal name for *var* and returning its value in a `finally` clause. The *var* argument is bound as if by the construct `with`. No mechanism is provided for declaring the *type* of *var*; it must be of *type* **list**.

The constructs `append`, `appending`, `nconc`, and `nconcing` are similar to `collect` except that the values of the supplied *form* must be *lists*.

- The `append` keyword causes its *list* values to be concatenated into a single *list*, as if they were arguments to the *function* **append**.

- The `nconc` keyword causes its *list* values to be concatenated into a single *list*, as if they were arguments to the *function* **nconc**.

The argument *var* is set to the *list* of concatenated values; if *var* is supplied, **loop** does not return the final *list* automatically. The *var* argument is bound as if by the construct `with`. A *type* cannot be supplied for *var*; it must be of *type* **list**. The construct `nconc` destructively modifies its argument *lists*.

The `count` construct counts the number of times that the supplied *form* returns *true*. The argument *var* accumulates the number of occurrences; if *var* is supplied, **loop** does not return the final count automatically. The *var* argument is bound as if by the construct `with`. If `into` *var* is used, a *type* can be supplied for *var* with the *type-spec* argument; the consequences are unspecified if a nonnumeric *type* is supplied. If there is no `into` variable, the optional *type-spec* argument applies to the internal variable that is keeping the count. The default *type* is *implementation-dependent*; but it must be a *subtype* of `(or integer float)`.

The `maximize` construct compares the value of the supplied *form* obtained during the first iteration with values obtained in successive iterations. The maximum value encountered is determined and returned. If **loop** never executes the body, the returned value is unspecified. The argument *var* accumulates the maximum or minimum value; if *var* is supplied, **loop** does not return the maximum or minimum automatically. The *var* argument is bound as if by the construct `with`. If `into` *var* is used, a *type* can be supplied for *var* with the *type-spec* argument; the consequences are unspecified if a nonnumeric *type* is supplied. If there is no `into` variable, the optional *type-spec* argument applies to the internal variable that is keeping the maximum or minimum value. The default *type* must be a *subtype* of `(or integer float)`.

The `minimize` construct is similar to `maximize`; it determines and returns the minimum value. The `minimize` construct compares the value of the supplied *form* obtained during the first iteration with values obtained in successive iterations. The minimum value encountered is determined and returned. If **loop** never iterates, the returned value is not meaningful. The argument *var* is bound to the minimum value; if *var* is supplied, the **loop** does not return the minimum automatically. The *var* argument is bound as if by the construct `with`. The *type-spec* argument supplies the *type* for *var*; the default type is **fixnum**. The consequences are unspecified if a nonnumeric *type* is supplied for *var*.

The `sum` construct forms a cumulative sum of the values of the supplied *form* at each iteration. The argument *var* is used to accumulate the sum; if *var* is supplied, **loop** does not return the final sum automatically. The *var* argument is bound as if by the construct `with`. If `into` *var* is used, a *type* can be supplied for *var* with the *type-spec* argument; the consequences are unspecified if a nonnumeric *type* is supplied. If there is no `into` variable, the optional *type-spec* argument applies to the internal variable that is keeping the sum. The default *type* must be a *subtype* of *type* **number**.

If `into` is used, the construct does not provide a default return value; however, the variable is available for use in any `finally` clause.

Value-returning accumulation clauses can be combined in a **loop** if all the clauses accumulate the same *type* of *object*. By default, the Loop Facility returns only one value; thus, the *objects* collected by multiple accumulation clauses as return values must have compatible *types*. For example, since both the `collect` and `append` constructs accumulate *objects* into a *list* that is returned from a **loop**, they can be combined safely.

```
;; Collect every name and the kids in one list by using
;; COLLECT and APPEND.
 (loop for name in '(fred sue alice joe june)
```

```
       for kids in '((bob ken) () () (kris sunshine) ())
       collect name
       append kids)
→ (FRED BOB KEN SUE ALICE JOE KRIS SUNSHINE JUNE)
```

Multiple clauses that do not accumulate the same *type* of *object* can coexist in a **loop** only if each clause accumulates its values into a different *variable*.

Local Variable Initializations

When a **loop** *form* is executed, the local variables are bound and are initialized to some value. These local variables exist until **loop** iteration terminates, at which point they cease to exist. Implicit variables are also established by iteration control clauses and the `into` preposition of accumulation clauses.

The `with` construct initializes variables that are local to a loop. The variables are initialized one time only. If the optional *type-spec* argument is supplied for the variable *var*, but there is no related expression to be evaluated, *var* is initialized to an appropriate default value for its *type*. For example, for the types **t**, **number**, and **float**, the default values are **nil**, `0`, and `0.0` respectively. The consequences are undefined if a *type-spec* argument is supplied for *var* if the related expression returns a value that is not of the supplied *type*. By default, the `with` construct initializes variables *sequentially*; that is, one variable is assigned a value before the next expression is evaluated. However, by using the *loop keyword* `and` to join several `with` clauses, initializations can be forced to occur in *parallel*; that is, all of the supplied *forms* are evaluated, and the results are bound to the respective variables simultaneously.

Sequential binding is used when it is desireable for the initialization of some variables to depend on the values of previously bound variables. For example, suppose the variables `a`, `b`, and `c` are to be bound in sequence:

```
 (loop with a = 1
       with b = (+ a 2)
       with c = (+ b 3)
       return (list a b c))
→ (1 3 6)
```

The execution of the above **loop** is equivalent to the execution of the following code:

```
 (let* ((a 1)
        (b (+ a 2))
        (c (+ b 3)))
   (block nil
     (tagbody
         (next-loop (return (list a b c))
                    (go next-loop)
                    end-loop))))
```

If the values of previously bound variables are not needed for the initialization of other local variables, an `and` clause can be used to specify that the bindings are to occur in *parallel*:

```
(loop with a = 1
      and b = 2
      and c = 3
      return (list a b c))
→ (1 2 3)
```

The execution of the above loop is equivalent to the execution of the following code:

```
(let ((a 1)
      (b 2)
      (c 3))
  (block nil
    (tagbody
        (next-loop (return (list a b c))
                   (go next-loop)
                   end-loop))))
```

Conditional Execution

The `if`, `when`, and `unless` constructs establish conditional control in a **loop**. If the supplied condition is *true*, the succeeding loop clause is executed. If the supplied condition is not true, the succeeding clause is skipped, and program control moves to the clause that follows the *loop keyword* `else`. If the supplied condition is not true and no `else` clause is supplied, control is transferred to the clause or construct following the supplied condition.

The constructs `if` and `when` allow execution of loop clauses conditionally. These constructs are synonyms and can be used interchangeably. If the value of the test expression *form* is *non-nil*, the expression *clause1* is evaluated. If the test expression evaluates to **nil** and an `else` construct is supplied, the *forms* that follow the `else` are evaluated; otherwise, control passes to the next clause. If `if` or `when` clauses are nested, each `else` is paired with the closest preceding `if` or `when` construct that has no associated `else` or `end`.

The `unless` construct is equivalent to `when (not` *form*`)` and `if (not` *form*`)`. If the value of the test expression *form* is **nil**, the expression *clause1* is evaluated. If the test expression evaluates to *non-nil* and an `else` construct is supplied, the statements that follow the `else` are evaluated; otherwise, no conditional statement is evaluated. The *clause* arguments must be either accumulation, unconditional, or conditional clauses.

Clauses that follow the test expression can be grouped by using the *loop keyword* `and` to produce a conditional block consisting of a compound clause.

The *loop keyword* `it` can be used to refer to the result of the test expression in a clause. If multiple clauses are connected with `and`, the `it` construct must be in the first clause in the block. Since `it` is a *loop keyword*, `it` cannot be used as a local variable within **loop**.

The optional *loop keyword* `end` marks the end of the clause. If this keyword is not supplied, the next *loop keyword* marks the end. The construct `end` can be used to distinguish the scoping of compound clauses.

Main Body

The `do` and `doing` constructs evaluate the supplied *forms* wherever they occur in the expanded form of **loop**. The *form* argument can be any *non-atomic form*. Each *form* is evaluated in every iteration. Because every loop clause must begin with a *loop keyword*, the keyword `do` is used when no control action other than execution is required.

The `return` construct takes one *form* and returns its *values*. It is equivalent to the clause `do (return` *form*`)`.

Initial and Final Evaluation

The `initially` and `finally` constructs evaluate forms that occur before and after the loop body.

The `initially` construct causes the supplied *form* to be evaluated in the loop prologue, which precedes all loop code except for initial settings supplied by constructs `with`, `for`, or `as`. The code for any `initially` clauses is executed in the order in which the clauses appeared in the **loop**. The *form* argument can be any nonatomic *form*.

The `finally` construct causes the supplied *form* to be evaluated in the loop epilogue after normal iteration terminates. The code for any `finally` clauses is executed in the order in which the clauses appeared in the **loop**. The collected code is executed once in the loop epilogue before any implicit values are returned from the accumulation clauses. An explicit transfer of control (*e.g.*, by **return**, **go**, or **throw**) from the loop body, however, will exit the **loop** without executing the epilogue code. The *form* argument can be any nonatomic *form*.

Clauses such as `return`, `always`, `never`, and `thereis` can bypass the `finally` clause. **return** (or **return-from**, if the `named` option was supplied) can be used after `finally` to return values from a **loop**. Such an *explicit return* inside the `finally` clause takes precedence over returning the accumulation from clauses supplied by such keywords as `collect`, `nconc`, `append`, `sum`, `count`, `maximize`, and `minimize`; the accumulation values for these preempted clauses are not returned by **loop** if **return** or **return-from** is used.

Miscellaneous Operations

The `named` construct establishes a name for an *implicit block* surrounding the **loop** so that the **return-from** *special operator* can be used to return values from or to exit **loop**. Only one name per **loop** *form* can be assigned. If used, the `named` construct must be the first clause in the loop expression.

The `return` construct terminates a loop and returns the value of the supplied *form* as the value of the **loop**. This construct is similar to the **return-from** *special operator* and the **return** *macro*. The `return` construct returns immediately and does not execute any `finally` clause that is given.

6.1.2.7 Order of Execution

With the exceptions listed below, clauses are executed in the loop body in the order in which they appear in the source. Execution is repeated until a clause terminates the **loop** or until a **return**, **go**, or **throw** form is encountered which transfers control to a point outside of the loop. The following actions are exceptions to the linear order of execution:

- All variables are initialized first, regardless of where the establishing clauses appear in the source. The order of initialization follows the order of these clauses.

- The code for any `initially` clauses is collected into one **progn** in the order in which the clauses appear in the source. The collected code is executed once in the loop prologue after any implicit variable initializations.

- The code for any `finally` clauses is collected into one **progn** in the order in which the clauses appear in the source. The collected code is executed once in the loop epilogue before any implicit values from the accumulation clauses are returned. Explicit returns anywhere in the source, however, will exit the **loop** without executing the epilogue code.

- A `with` clause introduces a variable *binding* and an optional initial value. The initial values are calculated in the order in which the `with` clauses occur.

- Iteration control clauses implicitly perform the following actions:

  - initialize variables;

  - *step* variables, generally between each execution of the loop body;

  - perform termination tests, generally just before the execution of the loop body.

6.1.2.8 Destructuring

The *d-type-spec* argument is used for destructuring. If the *d-type-spec* argument consists solely of the *type* **fixnum**, **float**, **t**, or **nil**, the `of-type` keyword is optional. The `of-type` construct is optional in these cases to provide backwards compatibility; thus, the following two expressions are the same:

```
;;; This expression uses the old syntax for type specifiers.
 (loop for i fixnum upfrom 3 ...)
```

```
;;; This expression uses the new syntax for type specifiers.
 (loop for i of-type fixnum upfrom 3 ...)

;; Declare X and Y to be of type VECTOR and FIXNUM respectively.
 (loop for (x y) of-type (vector fixnum)
       in l do ...)
```

A *type specifier* for a destructuring pattern is a *tree* of *type specifiers* with the same shape as the *tree* of *variable names*, with the following exceptions:

- When aligning the *trees*, an *atom* in the *tree* of *type specifiers* that matches a *cons* in the variable tree declares the same *type* for each variable in the subtree rooted at the *cons*.

- A *cons* in the *tree* of *type specifiers* that matches an *atom* in the *tree* of *variable names* is a *non-atomic type specifer*.

Destructuring allows *binding* of a set of variables to a corresponding set of values anywhere that a value can normally be bound to a single variable. During **loop** expansion, each variable in the variable list is matched with the values in the values list. If there are more variables in the variable list than there are values in the values list, the remaining variables are given a value of **nil**. If there are more values than variables listed, the extra values are discarded.

To assign values from a list to the variables `a`, `b`, and `c`, the `for` clause could be used to bind the variable `numlist` to the *car* of the supplied *form*, and then another `for` clause could be used to bind the variables `a`, `b`, and `c` *sequentially*.

```
;; Collect values by using FOR constructs.
 (loop for numlist in '((1 2 4.0) (5 6 8.3) (8 9 10.4))
       for a of-type integer = (first numlist)
       and b of-type integer = (second numlist)
       and c of-type float = (third numlist)
       collect (list c b a))
→ ((4.0 2 1) (8.3 6 5) (10.4 9 8))
```

Destructuring makes this process easier by allowing the variables to be bound in each loop iteration. *Types* can be declared by using a list of *type-spec* arguments. If all the *types* are the same, a shorthand destructuring syntax can be used, as the second example illustrates.

```
;; Destructuring simplifies the process.
 (loop for (a b c) of-type (integer integer float) in
       '((1 2 4.0) (5 6 8.3) (8 9 10.4))
       collect (list c b a))
→ ((4.0 2 1) (8.3 6 5) (10.4 9 8))


;; If all the types are the same, this way is even simpler.
```

```
 (loop for (a b c) of-type float in
       '((1.0 2.0 4.0) (5.0 6.0 8.3) (8.0 9.0 10.4))
       collect (list c b a))
→ ((4.0 2.0 1.0) (8.3 6.0 5.0) (10.4 9.0 8.0))
```

If destructuring is used to declare or initialize a number of groups of variables into *types*, the *loop keyword* `and` can be used to simplify the process further. `;; Initialize and declare variables in parallel by using the AND construct.`

```
 (loop with (a b) of-type float = '(1.0 2.0)
       and (c d) of-type integer = '(3 4)
       and (e f)
       return (list a b c d e f))
→ (1.0 2.0 3 4 NIL NIL)
```

If **nil** is used in a destructuring list, no variable is provided for its place.

```
 (loop for (a nil b) = '(1 2 3)
       do (return (list a b)))
→ (1 3)
```

Note that nonstandard lists can specify destructuring.

```
 (loop for (x . y) = '(1 . 2)
       do (return y))
→ 2
 (loop for ((a . b) (c . d)) of-type ((float . float) (integer . integer)) in
       '(((1.2 . 2.4) (3 . 4)) ((3.4 . 4.6) (5 . 6)))
       collect (list a b c d))
→ ((1.2 2.4 3 4) (3.4 4.6 5 6))
```

An error of *type* **program-error** is signaled (at macro expansion time) if the same variable is bound twice in any variable-binding clause of a single **loop** expression. Such variables include local variables, iteration control variables, and variables found by destructuring.

6.1.2.9 Restrictions on Side-Effects

See Section 3.6 (Traversal Rules and Side Effects).

6.1.3 Loop Examples

```
 (let ((i 0))                     ; no loop keywords are used
    (loop (incf i) (if (= i 3) (return i)))) → 3
 (let ((i 0)(j 0))
    (tagbody
      (loop (incf j 3) (incf i) (if (= i 3) (go exit)))
      exit)
```

```
    j)  → 9
```

In the following example, the variable `x` is stepped before `y` is stepped; thus, the value of `y` reflects the updated value of `x`:

```
 (loop for x from 1 to 10
       for y = nil then x
       collect (list x y))
→ ((1 NIL) (2 2) (3 3) (4 4) (5 5) (6 6) (7 7) (8 8) (9 9) (10 10))
```

In this example, `x` and `y` are stepped in *parallel*:

```
 (loop for x from 1 to 10
       and y = nil then x
       collect (list x y))
→ ((1 NIL) (2 1) (3 2) (4 3) (5 4) (6 5) (7 6) (8 7) (9 8) (10 9))
```

Examples of *for-as-arithmethic* subclause:

```
;; Print some numbers.
 (loop for i from 1 to 3
       do (print i))
▷ 1
▷ 2
▷ 3
→ NIL

;; Print every third number.
 (loop for i from 10 downto 1 by 3
       do (print i))
▷ 10
▷ 7
▷ 4
▷ 1
→ NIL

;; Step incrementally from the default starting value.
 (loop for i below 3
       do (print i))
▷ 0
▷ 1
▷ 2
→ NIL
```

Examples of *for-as-in-list* subclause:

```
;; Print every item in a list.
 (loop for item in '(1 2 3) do (print item))
```

```
▷ 1
▷ 2
▷ 3
→ NIL

;; Print every other item in a list.
 (loop for item in '(1 2 3 4 5) by #'cddr
       do (print item))
▷ 1
▷ 3
▷ 5
→ NIL

;; Destructure a list, and sum the x values using fixnum arithmetic.
 (loop for (item . x) (t . fixnum) in '((A . 1) (B . 2) (C . 3))
       unless (eq item 'B) sum x)
→ 4
```

Examples of *for-as-on-list* subclause:

```
;; Collect successive tails of a list.
 (loop for sublist on '(a b c d)
       collect sublist)
→ ((A B C D) (B C D) (C D) (D))

;; Print a list by using destructuring with the loop keyword ON.
 (loop for (item) on '(1 2 3)
       do (print item))
▷ 1
▷ 2
▷ 3
→ NIL
```

Example of *for-as-equals-then* subclause:

```
;; Collect some numbers.
 (loop for item = 1 then (+ item 10)
       for iteration from 1 to 5
       collect item)
→ (1 11 21 31 41)
```

Examples of *for-as-across* subclause:

```
 (loop for char across (the simple-string (find-message channel))
       do (write-char char stream))
```

Examples of *for-as-package* subclause:

```
 (let ((*package* (make-package "TEST-PACKAGE-1")))
   ;; For effect, intern some symbols
   (read-from-string "(THIS IS A TEST)")
   (export (intern "THIS"))
   (loop for x being each present-symbol of *package*
          do (print x)))
▷ A
▷ TEST
▷ THIS
▷ IS
→ NIL
```

Example of `repeat`**:**

```
 (loop repeat 3
       do (format t "~&What I say three times is true.~%"))
▷ What I say three times is true.
▷ What I say three times is true.
▷ What I say three times is true.
→ NIL
 (loop repeat -15
   do (format t "What you see is what you expect~%"))
→ NIL
```

Examples for `always`**,** `never`**, and** `thereis`**:**

```
;; Make sure I is always less than 11 (two ways).
;; The FOR construct terminates these loops.
 (loop for i from 0 to 10
       always (< i 11))
→ T
 (loop for i from 0 to 10
       never (> i 11))
→ T

;; If I exceeds 10 return I; otherwise, return NIL.
;; The THEREIS construct terminates this loop.
 (loop for i from 0
       thereis (when (> i 10) i) )
→ 11

;;; The FINALLY clause is not evaluated in these examples.
 (loop for i from 0 to 10
       always (< i 9)
```

```
       finally (print "you won't see this"))
→ NIL
 (loop never t
       finally (print "you won't see this"))
→ NIL
 (loop thereis "Here is my value"
       finally (print "you won't see this"))
→ "Here is my value"

;; The FOR construct terminates this loop, so the FINALLY clause
;; is evaluated.
 (loop for i from 1 to 10
       thereis (> i 11)
       finally (print i))
▷ 11
→ NIL

;; If this code could be used to find a counterexample to Fermat's
;; last theorem, it would still not return the value of the
;; counterexample because all of the THEREIS clauses in this example
;; only return T.  But if Fermat is right, that won't matter
;; because this won't terminate.

 (loop for z upfrom 2
       thereis
         (loop for n upfrom 3 below (log z 2)
               thereis
                 (loop for x below z
                       thereis
                         (loop for y below z
                               thereis (= (+ (expt x n) (expt y n))
                                          (expt z n))))))
```

Examples of `while` and `until`:

```
 (loop while (hungry-p) do (eat))

;; UNTIL NOT is equivalent to WHILE.
 (loop until (not (hungry-p)) do (eat))

;; Collect the length and the items of STACK.
 (let ((stack '(a b c d e f)))
   (loop while stack
         for item = (length stack) then (pop stack)
         collect item))
→ (6 A B C D E F)
```

```
;; Use WHILE to terminate a loop that otherwise wouldn't terminate.
;; Note that WHILE occurs after the WHEN.
 (loop for i fixnum from 3
       when (oddp i) collect i
       while (< i 5))
→ (3 5)
```

Examples of collect**:**

```
;; Collect all the symbols in a list.
 (loop for i in '(bird 3 4 turtle (1 . 4) horse cat)
       when (symbolp i) collect i)
→ (BIRD TURTLE HORSE CAT)

;; Collect and return odd numbers.
 (loop for i from 1 to 10
       if (oddp i) collect i)
→ (1 3 5 7 9)

;; Collect items into local variable, but don't return them.
 (loop for i in '(a b c d) by #'cddr
       collect i into my-list
       finally (print my-list))
▷ (A C)
→ NIL
```

Examples of append **and** nconc**:**

```
;; Use APPEND to concatenate some sublists.
  (loop for x in '((a) (b) ((c)))
        append x)
→ (A B (C))

;; NCONC some sublists together.  Note that only lists made by the
;; call to LIST are modified.
  (loop for i upfrom 0
        as x in '(a b (c))
        nconc (if (evenp i) (list x) nil))
→ (A (C))
```

Examples of count**:**

```
 (loop for i in '(a b nil c nil d e)
       count i)
→ 5
```

Examples of `maximize` and `minimize`:

```
 (loop for i in '(2 1 5 3 4)
       maximize i)
→ 5
 (loop for i in '(2 1 5 3 4)
       minimize i)
→ 1

;; In this example, FIXNUM applies to the internal variable that holds
;; the maximum value.
 (setq series '(1.2 4.3 5.7))
→ (1.2 4.3 5.7)
 (loop for v in series
       maximize (round v) of-type fixnum)
→ 6

;; In this example, FIXNUM applies to the variable RESULT.
 (loop for v of-type float in series
       minimize (round v) into result of-type fixnum
       finally (return result))
→ 1
```

Examples of `sum`:

```
 (loop for i of-type fixnum in '(1 2 3 4 5)
       sum i)
→ 15
 (setq series '(1.2 4.3 5.7))
→ (1.2 4.3 5.7)
 (loop for v in series
       sum (* 2.0 v))
→ 22.4
```

Examples of `when`:

```
;; Signal an exceptional condition.
 (loop for item in '(1 2 3 a 4 5)
       when (not (numberp item))
        return (cerror "enter new value" "non-numeric value: ~s" item))
Error: non-numeric value: A

;; The previous example is equivalent to the following one.
 (loop for item in '(1 2 3 a 4 5)
       when (not (numberp item))
        do (return
            (cerror "Enter new value" "non-numeric value: ~s" item)))
```

```
Error: non-numeric value: A

;; This example parses a simple printed string representation from
;; BUFFER (which is itself a string) and returns the index of the
;; closing double-quote character.
 (let ((buffer "\"a\" \"b\""))
   (loop initially (unless (char= (char buffer 0) #\")
                     (loop-finish))
         for i of-type fixnum from 1 below (length (the string buffer))
         when (char= (char buffer i) #\")
          return i))
→ 2

;; The FINALLY clause prints the last value of I.
;; The collected value is returned.
 (loop for i from 1 to 10
       when (> i 5)
         collect i
       finally (print i))
▷ 11
→ (6 7 8 9 10)

;; Return both the count of collected numbers and the numbers.
 (loop for i from 1 to 10
       when (> i 5)
         collect i into number-list
         and count i into number-count
       finally (return (values number-count number-list)))
→ 5, (6 7 8 9 10)
```

Examples of `named`**:**

```
;; Just name and return.
 (loop named max
       for i from 1 to 10
       do (print i)
       do (return-from max 'done))
▷ 1
→ DONE
```

Examples of *binding***:**

```
;; These bindings occur in sequence.
 (loop with a = 1
       with b = (+ a 2)
       with c = (+ b 3)
```

```
       return (list a b c))
→ (1 3 6)

;; These bindings occur in parallel.
 (setq a 5 b 10)
→ 10
 (loop with a = 1
       and b = (+ a 2)
       and c = (+ b 3)
       return (list a b c))
→ (1 7 13)

;; This example shows a shorthand way to declare local variables
;; that are of different types.
 (loop with (a b c) of-type (float integer float)
       return (format nil "~A ~A ~A" a b c))
→ "0.0 0 0.0"

;; This example shows a shorthand way to declare local variables
;; that are the same type.
 (loop with (a b c) of-type float
       return (format nil "~A ~A ~A" a b c))
→ "0.0 0.0 0.0"
```

Examples of clause grouping:

```
;; Group conditional clauses.
 (loop for i in '(1 324 2345 323 2 4 235 252)
       when (oddp i)
         do (print i)
         and collect i into odd-numbers
         and do (terpri)
       else                              ; I is even.
         collect i into even-numbers
       finally
         (return (values odd-numbers even-numbers)))
▷ 1
▷
▷ 2345
▷
▷ 323
▷
▷ 235
→ (1 2345 323 235), (324 2 4 252)

;; Collect numbers larger than 3.
```

```
 (loop for i in '(1 2 3 4 5 6)
       when (and (> i 3) i)
       collect it)                      ; IT refers to (and (> i 3) i).
→ (4 5 6)

;; Find a number in a list.
 (loop for i in '(1 2 3 4 5 6)
       when (and (> i 3) i)
       return it)
→ 4

;; The above example is similar to the following one.
 (loop for i in '(1 2 3 4 5 6)
       thereis (and (> i 3) i))
→ 4


;; Nest conditional clauses.
 (let ((list '(0 3.0 apple 4 5 9.8 orange marshmellow)))
   (loop for i in list
         when (numberp i)
           when (floatp i)
             collect i into float-numbers
           else                                  ; Not (floatp i)
             collect i into other-numbers
         else                                    ; Not (numberp i)
           when (symbolp i)
             collect i into symbol-list
           else                                  ; Not (symbolp i)
             do (error "found a funny value in list ~S, value ~S~%" list i)
         finally (return (values float-numbers other-numbers symbol-list))))
→ (3.0 9.8), (0 4 5), (APPLE ORANGE MARSHMELLOW)

;; Without the END preposition, the last AND would apply to the
;; inner IF rather than the outer one.
 (loop for x from 0 to 3
       do (print x)
       if (zerop (mod x 2))
         do (princ " a")
          and if (zerop (floor x 2))
                do (princ " b")
                end
          and do (princ " c"))
▷ 0  a b c
```

```
▷ 1
▷ 2  a c
▷ 3
→ NIL
```

Examples of unconditional execution:

```
;; Print numbers and their squares.
;; The DO construct applies to multiple forms.
 (loop for i from 1 to 3
       do (print i)
          (print (* i i)))
▷ 1
▷ 1
▷ 2
▷ 4
▷ 3
▷ 9
→ NIL
```

Notes:

Types can be supplied for loop variables. It is not necessary to supply a *type* for any variable, but supplying the *type* can ensure that the variable has a correctly typed initial value, and it can also enable compiler optimizations (depending on the implementation).

The clause `repeat` *n* ... is roughly equivalent to a clause such as

`(loop for` *internal-variable* `downfrom (-` *n* `1) to 0 ...)`

but in some implementations, the `repeat` construct might be more efficient.

Within the executable parts of the loop clauses and around the entire **loop** form, variables can be bound by using **let**.

There is *standardized* mechanism for users to add extensions to **loop**.

do, do* *Macro*

Syntax:

do ({*var* | (*var* [*init-form* [*step-form*]])}*)
(*end-test-form* {*result-form*}*)
{*declaration*}* {*tag* | *statement*}*

→ {*result*}*

do* ({*var* | (*var* [*init-form* [*step-form*]])}*)
(*end-test-form* {*result-form*}*)
{*declaration*}* {*tag* | *statement*}*

→ {*result*}*

Arguments and Values:

var—a *symbol*.

init-form—a *form*.

step-form—a *form*.

end-test-form—a *form*.

result-forms—an *implicit progn*.

declaration—a **declare** *expression*; not evaluated.

tag—a *go tag*; not evaluated.

statement—a *compound form*; evaluated as described below.

results—if a **return** or **return-from** form is executed, the *values* passed from that *form*; otherwise, the *values* returned by the *result-forms*.

Description:

do iterates over a group of *statements* while a test condition holds. **do** accepts an arbitrary number of iteration *vars* which are bound within the iteration and stepped in parallel. An initial value may be supplied for each iteration variable by use of an *init-form*. *Step-forms* may be used to specify how the *vars* should be updated on succeeding iterations through the loop. *Step-forms* may be used both to generate successive values or to accumulate results. If the *end-test-form* condition is met prior to an execution of the body, the iteration terminates. *Tags* label *statements*.

do* is exactly like **do** except that the *bindings* and steppings of the *vars* are performed sequentially rather than in parallel.

do, do*

Before the first iteration, all the *init-forms* are evaluated, and each *var* is bound to the value of its respective *init-form*, if supplied. This is a *binding*, not an assignment; when the loop terminates, the old values of those variables will be restored. For **do**, all of the *init-forms* are evaluated before any *var* is bound. The *init-forms* can refer to the *bindings* of the *vars* visible before beginning execution of **do**. For **do***, the first *init-form* is evaluated, then the first *var* is bound to that value, then the second *init-form* is evaluated, then the second *var* is bound, and so on; in general, the kth *init-form* can refer to the new binding of the jth *var* if $j < k$, and otherwise to the old binding of the jth *var*.

At the beginning of each iteration, after processing the variables, the *end-test-form* is evaluated. If the result is *false*, execution proceeds with the body of the **do** (or **do***) form. If the result is *true*, the *result-forms* are evaluated in order as an *implicit progn*, and then **do** or **do*** returns.

At the beginning of each iteration other than the first, *vars* are updated as follows. All the *step-forms*, if supplied, are evaluated, from left to right, and the resulting values are assigned to the respective *vars*. Any *var* that has no associated *step-form* is not assigned to. For **do**, all the *step-forms* are evaluated before any *var* is updated; the assignment of values to *vars* is done in parallel, as if by **psetq**. Because all of the *step-forms* are evaluated before any of the *vars* are altered, a *step-form* when evaluated always has access to the old values of all the *vars*, even if other *step-forms* precede it. For **do***, the first *step-form* is evaluated, then the value is assigned to the first *var*, then the second *step-form* is evaluated, then the value is assigned to the second *var*, and so on; the assignment of values to variables is done sequentially, as if by **setq**. For either **do** or **do***, after the *vars* have been updated, the *end-test-form* is evaluated as described above, and the iteration continues.

The remainder of the **do** (or **do***) form constitutes an *implicit tagbody*. *Tags* may appear within the body of a **do** loop for use by **go** statements appearing in the body (but such **go** statements may not appear in the variable specifiers, the *end-test-form*, or the *result-forms*). When the end of a **do** body is reached, the next iteration cycle (beginning with the evaluation of *step-forms*) occurs.

An *implicit block* named **nil** surrounds the entire **do** (or **do***) form. A **return** statement may be used at any point to exit the loop immediately.

Init-form is an initial value for the *var* with which it is associated. If *init-form* is omitted, the initial value of *var* is **nil**. If a *declaration* is supplied for a *var*, *init-form* must be consistent with the *declaration*.

Declarations can appear at the beginning of a **do** (or **do***) body. They apply to code in the **do** (or **do***) body, to the *bindings* of the **do** (or **do***) *vars*, to the *step-forms*, to the *end-test-form*, and to the *result-forms*.

Examples:

```
(do ((temp-one 1 (1+ temp-one))
      (temp-two 0 (1- temp-two)))
     ((> (- temp-one temp-two) 5) temp-one)) → 4
```

do, do*

```
(do ((temp-one 1 (1+ temp-one))
      (temp-two 0 (1+ temp-one)))
     ((= 3 temp-two) temp-one)) → 3

(do* ((temp-one 1 (1+ temp-one))
       (temp-two 0 (1+ temp-one)))
      ((= 3 temp-two) temp-one)) → 2

(do ((j 0 (+ j 1)))
    (nil)                       ;Do forever.
  (format t "~%Input ~D:" j)
  (let ((item (read)))
    (if (null item) (return)   ;Process items until NIL seen.
        (format t "~&Output ~D: ~S" j item))))
▷ Input 0: banana
▷ Output 0: BANANA
▷ Input 1: (57 boxes)
▷ Output 1: (57 BOXES)
▷ Input 2: NIL
→ NIL

(setq a-vector (vector 1 nil 3 nil))
(do ((i 0 (+ i 1))     ;Sets every null element of a-vector to zero.
     (n (array-dimension a-vector 0)))
    ((= i n))
  (when (null (aref a-vector i))
    (setf (aref a-vector i) 0))) → NIL
a-vector → #(1 0 3 0)

(do ((x e (cdr x))
     (oldx x x))
    ((null x))
  body)
```

is an example of parallel assignment to index variables. On the first iteration, the value of `oldx` is whatever value `x` had before the **do** was entered. On succeeding iterations, `oldx` contains the value that `x` had on the previous iteration.

```
(do ((x foo (cdr x))
     (y bar (cdr y))
     (z '() (cons (f (car x) (car y)) z)))
    ((or (null x) (null y))
     (nreverse z)))
```

does the same thing as `(mapcar #'f foo bar)`. The step computation for `z` is an example of the

do, do*

fact that variables are stepped in parallel. Also, the body of the loop is empty.

```
(defun list-reverse (list)
       (do ((x list (cdr x))
            (y '() (cons (car x) y)))
           ((endp x) y)))
```

As an example of nested iterations, consider a data structure that is a *list* of *conses*. The *car* of each *cons* is a *list* of *symbols*, and the *cdr* of each *cons* is a *list* of equal length containing corresponding values. Such a data structure is similar to an association list, but is divided into "frames"; the overall structure resembles a rib-cage. A lookup function on such a data structure might be:

```
(defun ribcage-lookup (sym ribcage)
       (do ((r ribcage (cdr r)))
           ((null r) nil)
         (do ((s (caar r) (cdr s))
              (v (cdar r) (cdr v)))
             ((null s))
           (when (eq (car s) sym)
             (return-from ribcage-lookup (car v)))))) → RIBCAGE-LOOKUP
```

See Also:

other iteration functions (**dolist**, **dotimes**, and **loop**) and more primitive functionality (**tagbody**, **go**, **block**, **return**, **let**, and **setq**)

Notes:

If *end-test-form* is **nil**, the test will never succeed. This provides an idiom for "do forever": the body of the **do** or **do*** is executed repeatedly. The infinite loop can be terminated by the use of **return**, **return-from**, **go** to an outer level, or **throw**.

A **do** *form* may be explained in terms of the more primitive *forms* **block**, **return**, **let**, **loop**, **tagbody**, and **psetq** as follows:

```
(block nil
  (let ((var1 init1)
        (var2 init2)
        ...
        (varn initn))
    declarations
    (loop (when end-test (return (progn . result)))
          (tagbody . tagbody)
          (psetq var1 step1
                 var2 step2
                 ...
                 varn stepn))))
```

do* is similar, except that **let*** and **setq** replace the **let** and **psetq**, respectively.

dotimes *Macro*

Syntax:

dotimes (*var count-form* [*result-form*]) {*declaration*}* {*tag* | *statement*}*
→ {*result*}*

Arguments and Values:

var—a *symbol*.

count-form—a *form*.

result-form—a *form*.

declaration—a **declare** *expression*; not evaluated.

tag—a *go tag*; not evaluated.

statement—a *compound form*; evaluated as described below.

results—if a **return** or **return-from** form is executed, the *values* passed from that *form*; otherwise, the *values* returned by the *result-form* or **nil** if there is no *result-form*.

Description:

dotimes iterates over a series of *integers*.

dotimes evaluates *count-form*, which should produce an *integer*. If *count-form* is zero or negative, the body is not executed. **dotimes** then executes the body once for each *integer* from 0 up to but not including the value of *count-form*, in the order in which the *tags* and *statements* occur, with *var* bound to each *integer*. Then *result-form* is evaluated. At the time *result-form* is processed, *var* is bound to the number of times the body was executed. *Tags* label *statements*.

In effect, a **block** named **nil** surrounds **dotimes**. **return** may be used to terminate the loop immediately without performing any further iterations, returning zero or more *values*.

The body of the loop is an *implicit tagbody*; it may contain tags to serve as the targets of **go** statements. Declarations may appear before the body of the loop.

For **dotimes**, the *scope* of the name binding does not include any initial value form, but the stepper and optional result forms are included.

Examples:

```
(dotimes (temp-one 10 temp-one)) → 10
(setq temp-two 0) → 0
```

```
 (dotimes (temp-one 10 t) (incf temp-two)) → T
 temp-two → 10
```

Here is an example of the use of `dotimes` in processing strings:

```
;;; True if the specified subsequence of the string is a
;;; palindrome (reads the same forwards and backwards).
 (defun palindromep (string &optional
                           (start 0)
                           (end (length string)))
   (dotimes (k (floor (- end start) 2) t)
    (unless (char-equal (char string (+ start k))
                        (char string (- end k 1)))
      (return nil))))
 (palindromep "Able was I ere I saw Elba") → T
 (palindromep "A man, a plan, a canal--Panama!") → NIL
 (remove-if-not #'alpha-char-p          ;Remove punctuation.
               "A man, a plan, a canal--Panama!")
→ "AmanaplanacanalPanama"
 (palindromep
  (remove-if-not #'alpha-char-p
                "A man, a plan, a canal--Panama!")) → T
 (palindromep
  (remove-if-not
   #'alpha-char-p
   "Unremarkable was I ere I saw Elba Kramer, nu?")) → T
 (palindromep
  (remove-if-not
   #'alpha-char-p
   "A man, a plan, a cat, a ham, a yak,
                  a yam, a hat, a canal--Panama!")) → T
```

See Also:

do, **dolist**, **tagbody**

Notes:

go may be used within the body of **dotimes** to transfer control to a statement labeled by a *tag*.

dolist *Macro*

Syntax:

dolist (*var list-form* [*result-form*]) {*declaration*}* {*tag* | *statement*}*
→ {*result*}*

dolist

Arguments and Values:

var—a *symbol*.

list-form—a *form*.

result-form—a *form*.

declaration—a **declare** *expression*; not evaluated.

tag—a *go tag*; not evaluated.

statement—a *compound form*; evaluated as described below.

results—if a **return** or **return-from** form is executed, the *values* passed from that *form*; otherwise, the *values* returned by the *result-form* or **nil** if there is no *result-form*.

Description:

dolist iterates over the elements of a *list*. The body of **dolist** is like a **tagbody**. It consists of a series of *tags* and *statements*.

dolist evaluates *list-form*, which should produce a *list*. It then executes the body once for each element in the *list*, in the order in which the *tags* and *statements* occur, with *var* bound to the element. Then *result-form* is evaluated. *tags* label *statements*.

At the time *result-form* is processed, *var* is bound to **nil**.

return may be used to terminate the loop and return a specified value.

In effect, a **block** named **nil** surrounds **dolist**.

For **dolist**, the *scope* of the name binding does not include any initial value form, but the stepper and optional result forms are included.

Examples:

```
 (setq temp-two '()) → NIL
 (dolist (temp-one '(1 2 3 4) temp-two) (push temp-one temp-two)) → (4 3 2 1)

 (setq temp-two 0) → 0
 (dolist (temp-one '(1 2 3 4)) (incf temp-two)) → NIL
 temp-two → 4

 (dolist (x '(a b c d)) (prin1 x) (princ " "))
▷ A B C D
→ NIL
```

See Also:

do, **dotimes**, **tagbody**, Section 3.6 (Traversal Rules and Side Effects)

Notes:

go may be used within the body of **dolist** to transfer control to a statement labeled by a *tag*.

loop *Macro*

Description:

For details, see Section 6.1 (The LOOP Iteration Facility).

Examples:

```
;; An example of the simple form of LOOP.
 (defun sqrt-advisor ()
   (loop
     (format t "~&Number: ")
     (let ((n (parse-integer (read-line) :junk-allowed t)))
       (when (not n) (return))
       (format t "~&The square root of ~D is ~D.~%"
               n (sqrt n)))))
→ SQRT-ADVISOR
 (sqrt-advisor)
▷ Number: 5↩
▷ The square root of 5 is 2.236068.
▷ Number: 4↩
▷ The square root of 4 is 2.
▷ Number: done↩
→ NIL

;; An example of the extended form of LOOP.
 (defun square-advisor ()
   (loop as n = (progn (format t "~&Number: ")
                       (parse-integer (read-line) :junk-allowed t))
         while n
         do (format t "~&The square of ~D is ~D.~%" n (* n n))))
→ SQUARE-ADVISOR
 (square-advisor)
▷ Number: 4↩
▷ The square of 4 is 16.
▷ Number: 23↩
▷ The square of 23 is 529.
▷ Number: done↩
→ NIL

;; Another example of the extended form of LOOP.
```

```
 (loop for n from 1 to 10
       when (oddp n)
         collect n)
→ (1 3 5 7 9)
```

Exceptional Situations:

None.

See Also:

do, **dolist**, **dotimes**, **return**, **go**, **throw**

Notes:

The simple form of **loop** is related to the extended form in the following way:

(loop {*compound-form*}*) ≡ (loop do {*compound-form*}*)

loop-finish *Local Macro*

Syntax:

(loop-finish) ⟨*no arguments*⟩

Description:

Can be used lexically within a **loop** *form* to terminate that *form* "normally." It transfers control to the loop epilogue, which permits execution of any **finally** clause (for effect) and then returns any accumulated result.

Examples:

```
;; Terminate the loop, but return the accumulated count.
 (loop for i in '(1 2 3 stop-here 4 5 6)
       when (symbolp i) do (loop-finish)
       count i)
→ 3

;; The preceding loop is equivalent to:
 (loop for i in '(1 2 3 stop-here 4 5 6)
       until (symbolp i)
       count i)
→ 3

;; While LOOP-FINISH can be used can be used in a variety of
;; situations it is really most needed in a situation where a need
```

loop-finish

```
;; to exit is detected at other than the loop's 'top level'
;; (where UNTIL or WHEN often work just as well), or where some
;; computation must occur between the point where a need to exit is
;; detected and the point where the exit actually occurs.  For example:
 (defun tokenize-sentence (string)
   (macrolet ((add-word (wvar svar)
                `(when ,wvar
                   (push (coerce (nreverse ,wvar) 'string) ,svar)
                   (setq ,wvar nil))))
     (loop with word = '() and sentence = '() and endpos = nil
           for i below (length string)
           do (let ((char (aref string i)))
                (case char
                  (#\Space (add-word word sentence))
                  (#\. (setq endpos (1+ i)) (loop-finish))
                  (otherwise (push char word))))
           finally (add-word word sentence)
                   (return (values (nreverse sentence) endpos)))))
→ TOKENIZE-SENTENCE

 (tokenize-sentence "this is a sentence. this is another sentence.")
→ ("this" "is" "a" "sentence"), 19

 (tokenize-sentence "this is a sentence")
→ ("this" "is" "a" "sentence"), NIL
```

Side Effects:

Transfers control.

Exceptional Situations:

Whether or not **loop-finish** is *fbound* in the *global environment* is *implementation-dependent*; however, the restrictions on redefinition and *shadowing* of **loop-finish** are the same as for *symbols* in the COMMON-LISP *package* which are *fbound* in the *global environment*. The consequences of attempting to use **loop-finish** outside of **loop** are undefined.

See Also:

loop

Notes:

Table of Contents