MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I.Memo No. 1236 August, 1991

The Iterate Manual

Jonathan Amsterdam

Abstract: This is the manual for version 1.1 of `iterate`, a powerful iteration macro for Common Lisp. `iterate` is similar to `loop` but provides numerous additional features, is well integrated with Lisp, and is extensible.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-85-K-0124.

Contents

1 Introduction **2**

2 Clauses **3**

2.1 Drivers 3

2.1.1 Numerical Iteration 4

2.1.2 Sequence Iteration 4

2.1.3 Generalized Drivers 6

2.1.4 Generators 7

2.1.5 Previous Values of Driver Variables 8

2.2 Variable Binding and Setting 9

2.3 Gathering Clauses 11

2.3.1 Reductions 11

2.3.2 Accumulations 12

2.3.3 Finders 13

2.3.4 Aggregated Boolean Tests 14

2.4 Control Flow 14

2.5 Code Placement 15

3 Other Features **16**

3.1 Multiple Accumulations 16

3.2 Named Blocks 16

3.3 Destructuring 17

3.4 On-line Help 18

3.5 Parallel Binding and Stepping 18

4 Types and Declarations **18**

4.1 Discussion 18

4.2 Summary 21

5 Problems with Code Movement **21**

6 Differences Between `Iterate` and `Loop` **22**

7 Rolling Your Own **23**

7.1 Introduction 23

7.2 Writing Drivers 26

7.3 Extensibility Aids 27

7.4 Subtleties 28

8 Obtaining `Iterate` **29**

1 Introduction

This manual describes `iterate`, a powerful iteration facility for Common Lisp. `iterate` provides abstractions for many common iteration patterns and allows for the definition of additional patterns. `iterate` is a macro that expands into ordinary Lisp at compile-time, so it is more efficient than higher-order functions like `map` and `reduce`. While it is similar to `loop`, `iterate` offers a more Lisp-like syntax and enhanced extensibility. (For a more complete comparison of `iterate` with other iteration constructs, see MIT AI Lab Working Paper No. 324, *Don't Loop, Iterate.*)

An `iterate` form consists of the symbol `iter`[1] followed by one or more forms, some of which may be `iterate` *clauses.* Here is a simple example of `iterate` which collects the numbers from 1 to 10 into a list, and returns the list. The return value is shown following the arrow.

```
(iter (for i from 1 to 10)
      (collect i))          ⇒ (1 2 3 4 5 6 7 8 9 10)
```

This form contains two clauses: a `for` clause that steps the variable `i` over the integers from 1 to 10, and a `collect` clause that accumulates its argument into a list. With a few exceptions, all `iterate` clauses have the same format: alternating symbols (called *keywords*) and expressions (called *arguments*). The syntax and terminology are those of Common Lisp's keyword lambda lists. One difference is that `iterate`'s keywords do not have to begin with a colon—though they may, except for the first symbol of a clause. So you can also write `(for i :from 1 :to 10)` if you prefer.

Any Lisp form can appear in the body of an `iterate`, where it will have its usual meaning. `iterate` walks the entire body, expanding macros, and recognizing clauses at any level. This example collects all the odd numbers in a list:

```
(iter (for el in list)
      (if (and (numberp el) (oddp el))
          (collect el)))
```

There are clauses for iterating over numbers, lists, arrays and other objects, and for collecting, summing, counting, maximizing and other useful operations. `iterate` also supports the creation of new variable bindings, stepping over multiple sequences at once, destructuring, and compiler declarations of variable types. The following example illustrates some of these features:

```
(iter (for (key . item) in alist)
         (for i from 0)
         (declare (fixnum i))
         (collect (cons i key)))
```

[1]You can also use `iterate`, but `iter` is preferred because it avoids potential conflicts with possible future additions to Common Lisp, and because it saves horizontal space when writing code.

This loop takes the keys of an alist and returns a new alist associating the keys with their positions in the original list. The compiler declaration for `i` will appear in the generated code in the appropriate place.

2 Clauses

Most of `iterate`'s clauses will be familiar to `loop` programmers. (`loop` is an iteration macro that has been incorporated into Common Lisp. Seen Guy Steele's *Common Lisp, 2nd Edition.*) In nearly all cases they behave the same as their `loop` counterparts, so a `loop` user can switch to `iterate` with little pain (and much gain).

All clauses with the standard keyword-argument syntax consist of two parts: a *required* part, containing keywords that must be present and in the right order; and an *optional* part, containing keywords that may be omitted and, if present, may occur in any order. In the descriptions below, the parts are separated by the Lisp lambda-list keyword `&optional`.

2.1 Drivers

An iteration-driving clause conceptually causes the iteration to go forward. Driver clauses in `iterate` allow iteration over numbers, lists, vectors, hashtables, packages, files and streams. Iteration-driving clauses must appear at the top level of an `iterate` form; they cannot be nested inside another clause. The driver variable is updated at the point where the driver clause occurs. Before the clause is executed for the first time, the value of the variable is undefined.

Multiple drivers may appear in a single `iterate` form, in which case all of the driver variables are updated each time through the loop, in the order in which the clauses appear. The first driver to terminate will terminate the entire loop.

In all cases, the value of the driver variable on exit from the loop, including within the epilogue code (see the `finally` clause), is undefined.

All the parameters of a driver clause are evaluated once, before the loop begins. Hence it is not possible to change the bounds or other properties of an iteration by side-effect from within the loop.

With one exception, driver clauses begin with the word `for` (or the synonym `as`) and mention an iteration variable, which is given a binding within the `iterate` form. The exception is `repeat`, which just executes a loop a specified number of times:

`repeat` *n*

Repeats the loop n times. For example:

```
(iter (repeat 100)
      (print "I will not talk in class."))
```

If $n \leq 0$, the loop will never be executed. If n is not an integer, the actual number of executions will be $\lceil n \rceil$.

2.1.1 Numerical Iteration

`for` *var* `&sequence`

The general form for iterating over a sequence of numbers requires a variable and, optionally, one or more keywords that provide the bounds and step size of the iteration. The `&sequence` lambda-list keyword is a shorthand for these sequence keywords. They are: `from`, `upfrom`, `downfrom`, `to`, `downto`, `above`, `below` and `by`. `from` provides the starting value for *var* and defaults to zero. `to` provides a final value and implies that the successive values of *var* will be increasing; `downto` implies that they will be decreasing. The loop terminates when *var* passes the final value (i.e. becomes smaller or larger than it, depending on the direction of iteration); in other words, the loop body will never be executed for values of *var* past the final value. `below` and `above` are similar to `to` and `downto`, except that the loop terminates when *var* equals or passes the final value.

If no final value is specified, the variable will be stepped forever. Using `from` or `upfrom` will result in increasing values, while `downfrom` will give decreasing values.

On each iteration, *var* is incremented or decremented by the value of the sequence keyword `by`, which defaults to 1. It should always be a positive number, even for downward iterations.

In the following examples, the sequence of numbers generated is shown next to the clause.

```
(for i upfrom 0) ⇒ 0 1 2 ...
(for i from 5)   ⇒ 5 6 7 ...     ; either from or upfrom is okay
(for i downfrom 0) ⇒ 0 -1 -2 ...
(for i from 1 to 3) ⇒ 1 2 3
(for i from 1 below 3) ⇒ 1 2
(for i from 1 to 3 by 2) ⇒ 1 3
(for i from 1 below 3 by 2) ⇒ 1
(for i from 5 downto 3) ⇒ 5 4 3
```

2.1.2 Sequence Iteration

There are a number of clauses for iterating over sequences. In all of them, the argument following `for` may be a list instead of a symbol, in which case destructuring is performed. See section 3.3.

`for` *var* `in` *list* `&optional by` *step-function*

var is set to successive elements of list. *step-function,* which defaults to `cdr`, is used to obtain the next sublist.

`for` *var* `on` *list* `&optional by` *step-function*

> *var* is set to successive sublists of list. *step-function* (default `cdr`) is used as in `for...in`.

These two clauses use `atom` to test for the end of a list. Hence, given a list whose final `cdr` is not `nil`, they will silently ignore the last `cdr`. Other choices are `endp`, which would signal an error, and `null`, which would probably result in an error somewhere else. If you wish to use an end-test other than `atom`, set the variable `iterate::*list-end-test*` to the name of the desired function.

`for` *var* `in-vector` *vector* `&sequence`

> *var* takes on successive elements from *vector.* The vector's fill-pointer is observed. Here and in subsequent clauses, the `&sequence` keywords include `with-index`, which takes a symbol as argument and uses it for the index variable instead of an internally generated symbol. The other `&sequence` keywords behave as in numerical iteration, except that the default iteration bounds are the bounds of the vector. E.g. in `(for i in-vector v downto 3)`, `i` will start off being bound to the last element in `v`, and will be set to preceding elements down to and including the element with index 3.

`for` *var* `in-sequence` *seq* `&sequence`

> This uses Common Lisp's generalized sequence functions, `elt` and `length`, to obtain elements and determine the length of *seq.* Hence it will work for any sequence, including lists, and will observe the fill-pointers of vectors.

`for` *var* `in-string` *string* `&sequence`

> *var* is set to successive characters of *string.*

`for` *var* `index-of-vector` *vector* `&sequence`
`for` *var* `index-of-sequence` *sequence* `&sequence`
`for` *var* `index-of-string` *string* `&sequence`

> *var* is set to successive indices of the sequence. These clauses avoid the overhead of accessing the sequence elements for those applications where they do not need to be examined, or are examined rarely. They admit all the optional keywords of the other sequence drivers except the (redundant) `with-index` keyword.

`for` *(key value)* `in-hashtable` *table*

> *key* and *value,* which must appear as shown in a list and may be destructuring templates, are set to the keys and values of *table.* If *key* is `nil`, then the hashtable's keys will be ignored; similarly for *value*. The order in which elements of *table* will be retrieved is unpredictable.

`for` *var* `in-package` *package* `&optional external-only` *ext*

Iterates over all the symbols in *package,* or over only the external symbols if *ext* is specified and non-`nil`. The same symbol may appear more than once.

Implementation note: Current versions of Common Lisp do not provide a mechanism that will allow `for`...`in-hashtable` and `for`...`in-package` to be implemented efficiently. However, the new Common Lisp standard does so, and future versions of `iterate` will take advantage of implementations that conform to the new standard.

`for` *var* `in-file` *name* `&optional using` *reader*

Opens the file *name* (which may be a string or pathname) for input, and iterates over its contents. *reader* defaults to `read`, so by default *var* will be bound to the successive forms in the file. The `iterate` body is wrapped in an unwind-protect to ensure that the file is closed no matter how the `iterate` is exited.

`for` *var* `in-stream` *stream* `&optional using` *reader*

Like `for`...`in-file`, except that *stream* should be an existing stream object that supports input operations.

2.1.3 Generalized Drivers

These are primarily useful for writing drivers that can also be used as generators (see section 2.1.4, below).

`for` *var* `next` *expr*

var is set to *expr* each time through the loop. Destructuring is performed. When the clause is used as a generator, *expr* is the code that is executed when `(next` *var*`)` is encountered (see section 2.1.4, below). *expr* should compute the first value for *var*, as well as all subsequent values, and is responsible for terminating the loop. For compatibility with future versions of `iterate`, this termination should be done with `terminate`, which can be considered a synonym for `finish` (see section 2.4).

As an example, the following clauses are equivalent to `(for i from 1 to 10)`:

```
(initially (setq i 0))
(for i next (if (> i 10) (terminate) (incf i)))
```

`for` *var* `do-next` *form*

form is evaluated each time through the loop. Its value is *not* set to *var*; that is *form's* job. *var* is only present so that `iterate` knows it is a driver variable. `(for` *var* `next` *expr*`)` is equivalent to `(for` *var* `do-next (dsetq` *var* *expr*`))`. (See section 3.3 for an explanation of `dsetq`.)

2.1.4 Generators

In all of the above clauses, the driver variable is updated on each iteration. Sometimes it is desirable to have greater control over updating. For instance, consider the problem of associating numbers, in increasing order and with no gaps, with the non-`nil` elements of a list. One obvious first pass at writing this is:

```
(iter (for el in list)
      (for i upfrom 1)
      (if el (collect (cons el i))))
```

But on the list `(a b nil c)` this produces `((a . 1) (b . 2) (c . 4))` instead of the desired `((a . 1) (b . 2) (c . 3))`. The problem is that `i` is incremented each time through the loop, even when `el` is `nil`.

The problem could be solved elegantly if we could step `i` only when we wished to. This can be accomplished for any `iterate` driver by writing `generate` (or its synonym `generating`) instead of `for`. Doing so produces a *generator*—a driver whose values are yielded explicitly. To obtain the next value of a generator variable v, write `(next` v`)`. The value of a `next` form is the next value of v, as determined by its associated driver clause. `next` also has the side-effect of updating v to that value. If there is no next value, `next` will terminate the loop, just as with a normal driver.

Using generators, we can now write our example like this:

```
(iter (for el in list)
      (generate i upfrom 1)
      (if el (collect (cons el (next i)))))
```

Now `i` is updated only when `(next i)` is executed, and this occurs only when `el` is non-`nil`.

To better understand the relationship between ordinary drivers and generators, observe that we can rewrite an ordinary driver using its generator form immediately followed by `next`, as this example shows:

```
(iter (generating i from 1 to 10)
      (next i)
      ...)
```

Provided that the loop body contains no `(next i)` forms, this will behave just as if we had written `(for i from 1 to 10)`.

We can still refer to a driver variable v without using `next`; in this case, its value is that given to it by the last evaluation of `(next` v`)`. Before `(next` v`)` has been called the first time, the value of v is undefined.

This semantics is more flexible than one in which *v* begins the loop bound to its first value and calls of `next` supply subsequent values, because it means the loop will not terminate too soon if the generator's sequence is empty. For instance, consider the following code, which tags non-`nil` elements of a list using a list of tags, and also counts the null elements. (We assume there are at least as many tags as non-`nil` elements.)

```
(let* ((counter 0)
       (tagged-list (iter (for el in list)
                          (generating tag in tag-list)
                          (if (null el)
                              (incf counter)
                              (collect (cons el (next tag)))))))
  ...)
```

It may be that there are just as many tags as non-null elements of `list`. If all the elements of `list` are null, we still want the counting to proceed, even though `tag-list` is `nil`. If `tag` had to be assigned its first value before the loop begins, we would have had to terminate the loop before the first iteration, since when `tag-list` is `nil`, `tag` has no first value. With the existing semantics, however, `(next tag)` will never execute, so the iteration will cover all the elements of `list`.

When the "variable" of a driver clause is actually a destructuring template containing several variables, all the variables are eligible for use with `next`. As before, `(next` *v*`)` evaluates to *v's* next value; but the effect is to update all of the template's variables. For instance, the following code will return the list `(a 2 c)`.

```
(iter (generating (key . item) in '((a . 1) (b . 2) (c . 3)))
      (collect (next key))
      (collect (next item)))
```

Only driver clauses with variables can be made into generators. This includes all clauses mentioned so far except for `repeat`. It does *not* include `for...previous`, `for...=`, `for...initially...then` or `for...first...then` (see below).

2.1.5 Previous Values of Driver Variables

Often one would like to access the value of a variable on a previous iteration. `iterate` provides a special clause for accomplishing this.

`for` *pvar* `previous` *var* `&optional initially` *init* `back` *n*

> Sets *pvar* to the previous value of *var,* which should be a driver variable, a variable from another `for...previous` clause, or a variable established by a `for...=`, `for...initially...then` or `for...first...then` clause (see section 2.2). Initially, *pvar* is given the value *init* (which defaults to `nil`). The *init* expression will be moved outside the loop body, so it should not depend on anything

computed within the loop. *pvar* retains the value of *init* until *var* is set to its second value, at which point *pvar* is set to *var's* first value; and so on.

The argument *n* to `back` must be a constant, positive integer, and defaults to 1. It determines how many iterations back *pvar* should track *var*. For example, when *n* is 2, then *pvar* will be assigned *var's* first value when *var* is set to its third value.

A `for...previous` clause may occur after or before its associated driver clause. `for...previous` works with generators as well as ordinary drivers.

Example:

```
(iter (for el in '(1 2 3 4))
      (for p-el previous el)
      (for pp-el previous p-el initially 0)
      (collect pp-el))
```

This evaluates to `(0 0 1 2)`. It could have been written more economically as

```
(iter (for el in '(1 2 3 4))
      (for pp-el previous el back 2 initially 0)
      (collect pp-el))
```

2.2 Variable Binding and Setting

Several clauses exist for establishing new variable bindings or for setting variables in the loop. They all support destructuring.

`with` *var* `&optional` `=` *value*

Causes *var* to be bound to value before the loop body is entered. If *value* is not supplied, *var* assumes a default binding, which will be `nil` in the absence of declarations. Also, if *value* is not supplied, no destructuring is performed; instead, *var* may be a list of symbols, all of which are given default bindings. If *value* is supplied, *var* is bound to it, with destructuring.

Because `with` Creates bindings whose scope includes the entire `iterate` form, it is good style to put all `with` clauses at the beginning.

Successive occurrences of `with` result in sequential bindings (as with `let*`). There is no way to obtain parallel bindings; see section 3.5 for a rationale.

`for` *var* `=` *expr*

On each iteration, *expr* is evaluated and *var* is set to its value.

This clause may appear to do the same thing as `for...next`. In fact, they are quite different. `for...=` provides only three services: it sets up a binding for *var*, sets it to *expr* on each iteration, and makes it possible to use `for...previous` with *var*. `for...next` provides these services in addition to the ability to turn the driver into a generator.

`for` *var* `initially` *init-expr* `then` *then-expr*

Before the loop begins, *var* is set to *init-expr;* on all iterations after the first it is set to *then-expr.* This clause must occur at top-level. *init-expr* will be moved outside the loop body and *then-expr* will be moved to the end of the loop body, so they are subject to code motion problems (see section 5).

This clause may appear to be similar to `for...next`, but in fact they differ significantly. `for...initially...then` is typically used to give *var* its first value before the loop begins, and subsequent values on following iterations. This is incompatible with generators, whose first value and subsequent values must all be computed by `(next` *var*`)`. Also, the update of *var* in `for...initially...then` does not occur at the location of the clause. Use `for...initially...then` for one-shot computations where its idiom is more convenient, but use `for...next` for extending `iterate` with new drivers (see section 7).

`for` *var* `first` *first-expr* `then` *then-expr*

The first time through the loop, *var* is set to *first-expr;* on subsequent iterations, it is set to *then-expr.* This differs from `for...initially` in that *var* is set to *first-expr* inside the loop body, so *first-expr* may depend on the results of other clauses. For instance,

```
(iter (for num in list)
      (for i first num then (1+ i))
      ...)
```

will set `i` to the first element of `list` on the first iteration, whereas

```
(iter (for num in list)
      (for i initially num then (1+ i))
      ...)
```

is probably erroneous; `i` will be bound to `num`'s default binding (usually `nil`) for the first iteration.

Compatibility note: `loop`'s `for...=` works like `iterate`'s, but `loop` used the syntax `for...=...then` to mean `for...initially...then`. It was felt that these two operations were sufficiently different to warrant different keywords.

Also, the `for` in the above three clauses is misleading, since none is true driver (e.g. none has a corresponding `generate` form). `setting` would have been a better choice, but `for` was used to retain some compatibility with `loop`.

2.3 Gathering Clauses

Many of `iterate`'s clauses accumulate values into a variable, or set a variable under certain conditions. At the end of the loop, this variable contains the desired result. All these clauses have an optional `into` keyword, whose argument should be a symbol. If the `into` keyword is not supplied, the accumulation variable will be internally generated and its value will be returned at the end of the loop; if a variable is specified, that variable is used for the accumulation, and is not returned as a result—it is up to the user to return it explicitly, in the loop's epilogue code (see `finally`). It is safe to examine the accumulation variable during the loop, but it should not be modified.

These clauses all begin with a verb. When the verb does not conflict with an existing Common Lisp function, then it may be used in either its infinitival or present-participle form (e.g. `sum`, `summing`). However, when there is a conflict with Common Lisp, only the present-participle form may be used (e.g. `unioning`). This is to prevent `iterate` clauses from clashing with Common Lisp functions.

2.3.1 Reductions

Reduction is an extremely common iteration pattern in which the results of successive applications of a binary operation are accumulated. For example, a loop that computes the sum of the elements of a list is performing a reduction with the addition operation. This could be written in Common Lisp as `(reduce #'+ list)` or with `iterate` as

```
(iter (for el in list)
      (sum el))
```

`sum` *expr* `&optional into` *var*

Each time through the loop, *expr* is evaluated and added to a variable, which is bound initially to zero. If *expr* has a type, it is *not* used as the type of the sum variable, which is always `number`. To get the result variable to be of a more specific type, use an explicit variable, as in

```
(iter (for el in number-list)
      (sum el into x)
      (declare (fixnum x))
      (finally (return x)))
```

`multiply` *expr* `&optional into` *var*

Like `sum`, but the initial value of the result variable is 1, and the variable is updated by multiplying *expr* into it.

`counting` *expr* `&optional into` *var*

expr is evaluated on each iteration. If it is non-`nil`, the accumulation variable, initially zero, is incremented.

`maximize` *expr* `&optional into` *var*
`minimize` *expr* `&optional into` *var*

expr is evaluated on each iteration and its extremum (maximum or minimum) is stored in the accumulation variable. If *expr* is never evaluated, then the result is `nil` (if the accumulation variable is untyped) or 0 (if it has a numeric type).

`reducing` *expr* `by` *func* `&optional initial-value` *init-val* `into` *var*

This is a general way to perform reductions. *func* should be a function of two arguments, the first of which will be the value computed so far and the second of which will be the value of *expr*. It should return the new value. `reducing` is roughly equivalent to the Common Lisp `(reduce` *func list*`)`, where *list* is a list of the successive values of *expr*. Because this list need not be explicitly constructed, the `iterate` clause may be more efficient than the Common Lisp function.

If the `reducing` clause is never executed, the result is undefined.

It is not necessary to provide an initial value, but better code can be generated if one is supplied. Regardless of its location in the `iterate` body, *init-val* will be evaluated before the loop is entered, so it should not depend on any value computed inside the `iterate` form.

2.3.2 Accumulations

All the predefined accumulation clauses add values to a sequence. If the sequence is a list, they all have the property that the partial list is kept in the correct order and available for inspection at any point in the loop.

`collect` *expr* `&optional into` *var* `at` *place* `result-type` *type*

Produces a sequence of the values of *expr* on each iteration. *place* indicates where the next value of *expr* is added to the list and may be one of the symbols `start`, `beginning` (a synonym for `start`) or `end`. The symbol may be quoted, but need not be. The default is `end`. For example,

```
(iter (for i from 1 to 5)
      (collect i))
```

produces `(1 2 3 4 5)`, whereas

```
(iter (for i from 1 to 5)
      (collect i at beginning))
```

produces `(5 4 3 2 1)` (and is likely to be faster in most Common Lisp implementations).

If *type* is provided, it should be a subtype of `sequence`. The default is `list`. Specifying a type other than `list` will result in `collect` returning a sequence of that type. *However,* the type of the sequence being constructed when inside the loop body is undefined when a non-`list` type is specified. (As with *place*, quoting *type* is optional.)

`adjoining` *expr* `&optional into` *var* `test` *test* `at` *place* `result-type` *type*

Like `collect`, but only adds the value of *expr* if it is not already present. *test,* which defaults to `#'eql`, is the test to be used with `member`.

`appending` *expr* `&optional into` *var* `at` *place*
`nconcing` *expr* `&optional into` *var* `at` *place*
`unioning` *expr* `&optional into` *var* `at` *place*
`nunioning` *expr* `&optional into` *var* `at` *place*

These are like `collect`, but behave like the Common Lisp functions `append`, `nconc`, `union` or `nunion`. As in Common Lisp, they work only on lists. Also as in Common Lisp, `unioning` and `nunioning` assume that the value of *expr* contains no duplicates.

`accumulate` *expr* `by` *func* `&optional initial-value` *init-val* `into` *var*

This is a general-purpose accumulation clause. *func* should be a function of two arguments, the value of *expr* and the value accumulated so far in the iteration, and it should return the updated value. If no initial value is supplied, `nil` is used.

The differences between `accumulate` and `reducing` are slight. One difference is that the functions take their arguments in a different order. Another is that in the absence of *init-val*, `accumulate` will use `nil`, whereas `reducing` will generate different code that avoids any dependence on the initial value. The reason for having both clauses is that one usually thinks of reductions (like `sum`) and accumulations (like `collect`) as different beasts.

2.3.3 Finders

A *finder* is a clause whose value is an expression that meets some condition.

`finding` *expr* `such-that` *test* `&optional into` *var* `on-failure` *failure-value*

If *test* (which is an expression) ever evaluates to non-`nil`, the loop is terminated, the epilogue code is run and the value of *expr* is returned. Otherwise, `nil` (or *failure-value,* if provided) is returned. If *var* is provided, it will have either the non-`nil` value of *expr* or *failure-value* when the epilogue code is run.

As a special case, if the *test* expression is a sharp-quoted function, then it is applied to *expr* instead of being simply evaluated. E.g. `(finding x such-that #'evenp)` is equivalent to `(finding x such-that (evenp x))`.

`finding` *expr* `maximizing` *m-expr* `&optional into` *var*
`finding` *expr* `minimizing` *m-expr* `&optional into` *var*

Computes the extremum (maximum or minimum) value of *m-expr* over all iterations, and returns the value of *expr* corresponding to the extremum. *expr* is evaluated inside the loop at the time the new extremum is established. If *m-expr* is never evaluated (due to, for example, being embedded in a conditional clause), then the returned value depends on the type, if any, of *expr* (or *var,* if one is supplied). If there is no type, the returned value will be `nil`; if the type is numeric, the returned value will be zero.

For these two clauses, *var* may be a list of two symbols; in that case, the first is used to record *expr* and the second, *m-expr.*

As with `finding...such-that`, if *m-expr* is a sharp-quoted function, then it is called on *expr* instead of being evaluated.

2.3.4 Aggregated Boolean Tests

`always` *expr*

If *expr* ever evaluates to `nil`, then `nil` is immediately returned; the epilogue code is not executed. If *expr* never evaluates to `nil`, the epilogue code is executed and the last value of *expr* (or `t` if *expr* was never evaluated) is returned.

`never` *expr*

Equivalent to `(always (not` *expr*`))`.

`thereis` *expr*

If *expr* is ever non-`nil`, its value is immediately returned without running epilogue code. Otherwise, the epilogue code is performed and `nil` is returned.

2.4 Control Flow

Several clauses can be used to alter the usual flow of control in a loop.

Note: the clauses of this and subsequent sections don't adhere to `iterate`'s usual syntax, but instead use standard Common Lisp syntax. Hence the format for describing syntax subsequently is like the standard format used in the Common Lisp manual, not like the descriptions of clauses above.

`finish`

Stops the loop and runs the epilogue code.

`leave &optional` *value*

Immediately returns *value* (default `nil`) from the current `iterate` form, skipping the epilogue code.

`next-iteration`

Skips the remainder of the loop body and begins the next iteration of the loop.

`while` *expr*

If *expr* ever evaluates to `nil`, the loop is terminated and the epilogue code executed. equivalent to `(if (not` *expr*`) (finish))`.

`until` *expr*

Equivalent to `(if` *expr* `(finish))`.

`if-first-time` *then* `&optional` *else*

If this clause is being executed for the first time in this invocation of the `iterate` form, then the *then* code is evaluated; otherwise the *else* code is evaluated.

`(for` *var* `first` *expr1* `then` *expr2*`)` is almost equivalent to

```
(if-first-time (dsetq var expr1)
               (dsetq var expr2))
```

The only difference is that the `for` version makes *var* available for use with `for...previous`.

2.5 Code Placement

When fine control is desired over where code appears in a loop generated by `iterate`, the following special clauses may be useful. They are all subject to code-motion problems (see section 5).

`initially &rest` *forms*

The lisp *forms* are placed in the prologue section of the loop, where they are executed once, before the loop body is entered.

`after-each &rest` *forms*

The *forms* are placed at the end of the loop body, where they are executed after each iteration. Unlike the other clauses in this section, *forms* may contain `iterate` clauses.

`finally &rest` *forms*

The lisp *forms* are placed in the epilogue section of the loop, where they are executed after the loop has terminated normally.

`finally-protected &rest` *forms*

The lisp *forms* are placed in the second form of an unwind-protect outside the loop. They are always executed after the loop has terminated, regardless of how the termination occurred.

3 Other Features

3.1 Multiple Accumulations

It is permitted to have more than one clause accumulate into the same variable, as in the following:

```
(iter (for i from 1 to 10)
      (collect i into nums)
      (collect (sqrt i) into nums)
      (finally (return nums)))
```

Clauses can only accumulate into the same variable if they are compatible. `collect`, `adjoining`, `appending`, `nconcing`, `unioning` and `nunioning` are compatible with each other; `sum`, `multiply` and `counting` are compatible; and `maximize` and `minimize` clauses are compatible only with other `maximize` and `minimize` clauses, respectively.

3.2 Named Blocks

Like Common Lisp `blocks`, `iterate` forms can be given names. The name should be a single symbol, and it must be the first form in the `iterate`. The generated code behaves exactly like a named block; in particular, `(return-from` *name*`)` can be used to exit it:

```
(iter fred
      (for i from 1 to 10)
      (iter barney
            (for j from i to 10)
            (if (> (* i j) 17)
                (return-from fred j))))
```

An `iterate` form that is not given a name is implicitly named `nil`.

Sometimes one would like to write an expression in an inner `iterate` form, but have it processed by an outer `iterate` form. This is possible with the `in` clause.

`in` *name* `&rest` *forms*

Evaluates *forms* as if they were part of the `iterate` form named *name*. In other words, `iterate` clauses are processed by the `iterate` form named *name,* and not by any `iterate` forms that occur inside *name.*

As an example, consider the problem of collecting a list of the elements in a two-dimensional array. The naive solution,

```
(iter (for i below (array-dimension ar 0))
      (iter (for j below (array-dimension ar 1))
            (collect (aref ar i j))))
```

is wrong because the list created by the inner `iterate` is simply ignored by the outer one. But using `in` we can write:

```
(iter outer (for i below (array-dimension ar 0))
      (iter (for j below (array-dimension ar 1))
            (in outer (collect (aref ar i j)))))
```

which has the desired result.

3.3 Destructuring

In many places within `iterate` clauses where a variable is expected, a list can be written instead. In these cases, the value to be assigned is *destructured* according to the pattern described by the list. As a simple example, the clause

```
(for (key . item) in alist)
```

will result in `key` being set to the `car` of each element in `alist`, and `item` being set to the `cdr`. The pattern list may be nested to arbitrary depth, and (as the example shows) need not be terminated with `nil`; the only requirement is that each leaf be a bindable symbol (or `nil`, in which case no binding is generated for that piece of the structure).

Sometimes, you might like to do the equivalent of a `multiple-value-setq` in a clause. This "multiple-value destructuring" can be expressed by writing `(values` pat_1 pat_2 `...)` for a destructuring pattern, as in

```
(for (values (a . b) c d) = (three-valued-function ...))
```

Note that the pat_i can themselves be destructuring patterns (though not multiple-value destructuring patterns). You can't do multiple-value destructuring in a `with` clause; instead wrap the whole `iterate` form in a `multiple-value-bind`.

Rationale: There are subtle interactions between variable declarations and evaluation order that make the correct implementation of multiple-value destructuring in a `with` somewhat tricky.

The destructuring feature of `iterate` is available as a separate mechanism, using the `dsetq` macro:

`dsetq` *template expr* [*Macro*]

Performs destructuring of *expr* using *template*. May be used outside of an `iterate` form.

3.4 On-line Help

There is a limited facility for on-line help, in the form of the `display-iterate-clauses` function.

`display-iterate-clauses &optional` *clause-spec* [*Function*]

Displays a list of `iterate` clauses. If *clause-spec* is not provided, all clauses are shown; if it is a symbol, all clauses beginning with that symbol are shown; and if it is a list of symbols, all clauses for which *clause-spec* is a prefix are shown.

3.5 Parallel Binding and Stepping

The parallel binding and stepping of variables is a feature that `iterate` does *not* have. This section attempts to provide a rationale.

We say that two variables are bound *in parallel* if neither binding shadows the other. This is the usual semantics of `let` (as opposed to `let*`). Similarly, we can say that iteration variables are stepped in parallel if neither variable is updated before the other, conceptually speaking; in other words, if the code to update each variable can reference the old values of both variables.

`loop` allows parallel binding of variables and parallel stepping of driver variables. My view is that if you are depending on the serial/parallel distinction, you are doing something obscure. If you need to bind variables in parallel using `with`, then you must be using a variable name that shadows a name in the existing lexical environment. Don't do that. The most common use for parallel stepping is to track the values of variables on the previous iteration, but in fact this does not require parallel stepping at all; the following will work:

```
(iter (for current in list)
      (for prev previous current)
      ...)
```

4 Types and Declarations

4.1 Discussion

Sometimes efficiency dictates that the types of variables be declared. This type information needs to be communicated to `iterate` so it can bind variables to appropriate values. Furthermore, `iterate` must often generate internal variables invisible to the user; there needs to be a way for these to be declared.

As an example, consider this code, which will return the number of odd elements in `number-list`:

```
(iter (for el in number-list)
      (count (oddp el)))
```

In processing this form, `iterate` will create an internal variable, let us call it `list17`, to hold the successive `cdr`s of `number-list`, and will bind the variable to `number-list`. It will also generate a default binding for `el`; only inside the body of the loop will `el` be set to the `car` of `list17`. Finally, `iterate` will generate a variable, call it `result`, to hold the result of the count, and will bind it to zero.

When dealing with type declarations, `iterate` observes one simple rule: *it will never generate a declaration unless requested to do so.* The reason is that such declarations might mask errors in compiled code by avoiding error-checks; the resulting problems would be doubly hard to track down because the declarations would be hidden from the programmer. Of course, a compiler might omit error-checks even in the absence of declarations, though this behavior can usually be avoided, e.g. by saying `(proclaim '(optimize (safety 3)))`.

So, the above `iterate` form will generate code with no declarations. But say we wish to declare the types of `el` and the internal variables `list17` and `result`. How is this done?

Declaring the type of `el` is easy, since the programmer knows the variable's name:

```
(iter (for el in number-list)
      (declare (fixnum el))
      (counting (oddp el)))
```

`iterate` can read variable type declarations like this one. Before processing any clauses, it scans the entire top-level form for type declarations and records the types, so that variable bindings can be performed correctly. In this case, `el` will be bound to zero instead of `nil`. Also, `iterate` collects all the top-level declarations and puts them at the begining of the generated code, so it is not necessary to place all declarations at the beginning of an `iterate` form; instead, they can be written near the variables whose types they declare.

Since `iterate` is not part of the compiler, it will not know about declarations that occur outside an `iterate` form; these declarations must be repeated inside the form.

Here is another way we could have declared the type of `el`:

```
(iter (for (the fixnum el) in number-list)
      (counting (oddp el)))
```

`iterate` extends the Common Lisp `the` form to apply to variables as well as value-producing forms; anywhere a variable is allowed—in a `with` clause, as the iteration variable in a driver clause, as the `into` argument of an accumulation clause, even inside a destructuring template—you can write `(the` *type symbol*`)` instead.

There is one crucial difference between using a `the` form and actually declaring the variable: explicit declarations are always placed in the generated code, but type information from a `the` form is not turned into an actual declaration unless you tell `iterate` to do so using `iterate::declare-variables`. See below.

Declaring the types of internal variables is harder than declaring the types of explicitly mentioned variables, since their names are unknown. You do it by declaring `iterate::declare-variables` somewhere inside the top level of the `iterate` form. (This will also generate declarations for variables declared using `the`.) `iterate` does not provide much selectivity here: it's all or none. And unfortunately, since `iterate` is not privy to compiler information but instead reads declarations itself, it will not hear if you `(proclaim '(iterate::declare-variables))`. Instead, set the variable `iterate::*always-declare-variables*` to `t` at compile-time, using `eval-when`.

To determine the appropriate types for internal variables, `iterate` uses three sources of information:

- Often, the particular clause dictates a certain type for a variable; `iterate` will use this information when available. In the current example, the variable `list17` will be given the type `list`, since that is the only type that makes sense; and the variable `result` will be given the type `fixnum`, on the assumption that you will not be counting high enough to need bignums. You can override this assumption only by using and explicitly declaring a variable:

  ```
  (iter (declare (iterate::declare-variables))
        (for el in number-list)
        (count (oddp el) into my-result)
        (declare (integer my-result))
        (finally (return my-result)))
  ```

  Other examples of the type assumptions that `iterate` makes are: type `list` for `into` variables of collection clauses; type `list` for expressions that are to be destructured; type `vector` for the variable holding the vector in a `for`...`in-vector` clause, and similarly for `string` and the `for`...`in-string` clause; and the implementation-dependent type `(type-of array-dimension-limit)` for the index and limit variables generated by sequence iteration drivers like `for`...`in-vector` and `for`...`in-string` (but not `for`...`in-sequence`, because it may be used to iterate over a list).

- Sometimes, `iterate` will examine expressions and try to determine their types in a simple-minded way. If the expression is self-evaluating (like a number, for instance), `iterate` knows that the expression's type is the same as the type of the value it denotes, so it can use that type. If the expression is of the form `(the` *type expr*`)`, `iterate` is smart enough to extract *type* and use it. However, the current version of `iterate` does not examine declarations of function result types or do any type inference. It will not determine, for example, that the type of `(+ 3 4)` is `fixnum`, or even `number`.

- In some cases, the type of an internal variable should match the type of some other variable. For instance, `iterate` generates an internal variable for `(f x)` in the clause `(for i from 1 to (f x))`, and in the absence of other information will give it the same type as `i`. If, however, the expression had been written `(the fixnum (f x))`, then `iterate` would have given the internal variable the type `fixnum` regardless of `i`'s type. The type incompatibility errors that could arise in this situation are not checked for.

Note that if you do declare `iterate::declare-variables`, then `iterate` may declare user variables as well as internal ones if they do not already have declarations, though only for variables that it binds. For instance, in this code:

```
(iter (declare (iterate::declare-variables))
      (for i from 1 to 10)
      (collect i into var))
```

the variable `var` will be declared to be of type `list`.

4.2 Summary

`iterate` understands standard Common Lisp variable type declarations that occur within an `iterate` form and will pass them through to the generated code. If the declaration `(iterate::declare-variables)` appears at the top level of an `iterate` form, or if `iterate::*always-declare-variables*` is non-`nil`, then `iterate` will use the type information gleaned from user declarations, self-evaluating expressions and `the` expressions, combined with reasonable assumptions, to determine variable types and declare them.

5 Problems with Code Movement

Some `iterate` clauses, or parts of clauses, result in code being moved from the location of the clause to other parts of the loop. Drivers behave this way, as do code-placement clauses like `initially` and `finally`. When using these clauses, there is a danger of writing an expression that makes sense in its apparent location but will be invalid or have a different meaning in another location. For example:

```
(iter (for i from 1 to 10)
      (let ((x 3))
        (initially (setq x 4))))
```

While it may appear that the `x` of `(initially (setq x 4))` is the same as the `x` of `(let ((x 3))` ..., in fact they are not: `initially` moves its code outside the loop body, so `x` would refer to a global variable. Here is another example of the same problem:

```
(iter (for i from 1 to 10)
      (let ((x 3))
        (collect i into x)))
```

If this code were executed, `collect` would create a binding for its `x` at the top level of the `iterate` form that the `let` will shadow.

Happily, `iterate` is smart enough to catch these errors; it walks all problematical code to ensure that free variables are not bound inside the loop body, and checks all variables it binds for the same problem.

However, some errors cannot be caught:

```
(iter (with x = 3)
      (for el in list)
      (setq x 1)
      (reducing el by #'+ initial-value x))
```

`reducing` moves its `initial-value` argument to the initialization part of the loop in order to produce more efficient code. Since `iterate` does not perform data-flow analysis, it cannot determine that `x` is changed inside the loop; all it can establish is that `x` is not bound internally. Hence this code will not signal an error and will use 3 as the initial value of the reduction.

The following list summarizes all cases that are subject to these code motion and variable-shadowing problems.

- Any variable for which `iterate` creates a binding, including those used in `with` and the `into` keyword of many clauses.
- The special clauses which place code: `initially`, `after-each`, `finally` and `finally-protected`.
- The variable of a `next` form.
- The `initially` arguments of `for...initially...then` and `for...previous`.
- The `then` argument of `for...initially...then`.
- The `initial-value` arguments of `reducing` and `accumulate`.
- The `on-failure` argument of `finding...such-that`.

6 Differences Between `Iterate` and `Loop`

`loop` contains a great deal of complexity which `iterate` tries to avoid. Hence many esoteric features of `loop` don't exist in `iterate`. Other features have been carried over, but in a cleaned-up form. And of course, many new features have been added; they are not mentioned in this list.

- `iterate`'s syntax is more Lisp-like than `loop`'s, having a higher density of parens.
- The current implementation of `iterate`, unlike the current version of `loop` (as documented in *Common Lisp, 2nd Ed.*), is extensible (see section 7).
- `loop` puts the updates of all driver variables at the top of the loop; `iterate` leaves them where the driver clauses appear.
- While for the most part `iterate` clauses that resemble `loop` clauses behave similarly, there are some differences. For instance, there is no `for`...`=`...`then` in `iterate`; instead use `for`...`initially`...`then`.
- `loop` binds the variable `it` at certain times to allow pseudo-English expressions like `when` *expr* `return it`. In `iterate`, you must bind *expr* to a variable yourself.
- `loop` has a special `return` clause, illustrated in the previous item. `iterate` doesn't need one, since an ordinary Lisp `return` has the same effect.
- `loop` allows for parallel binding and stepping of iteration variables. `iterate` does not. (See section 3.5.)
- `loop` and `iterate` handle variable type declarations very differently. `loop` provides a special syntax for declaring variable types, and does not examine declarations. Moreover, the standard implementation of `loop` will generate declarations when none are requested. `iterate` parses standard Common Lisp type declarations, and will never declare a variable itself unless declarations are specifically requested.

7 Rolling Your Own

7.1 Introduction

`iterate` is extensible—you can write new clauses that embody new iteration patterns. You might want to write a new driver clause for a data structure of your own, or you might want to write a clause that collects or manipulates elements in a way not provided by `iterate`.

This section describes how to write clauses for `iterate`. Writing a clause is like writing a macro. In fact, writing a clause *is* writing a macro: since `iterate` code-walks its body and macroexpands, you can add new abstractions to `iterate` with good old `defmacro`.

Actually, there are two extensions you can make to `iterate` that are even easier than writing a macro. They are adding a synonym for an existing clause and defining a driver clause for an indexable sequence. These can be done with `defsynonym` and `defclause-sequence`, respectively. See section 7.3, below.

The rest of this section explains how to write macros that expand into `iterate` clauses. Here's how you could add a simplified version of `iterate`'s `multiply` clause, if `iterate` didn't already have one:

```
(defmacro multiply (expr)
  `(reducing ,expr by #'* initial-value 1))
```

If you found yourself summing the square of an expression often, you might want to write a macro for that. A first cut might be

```
(defmacro sum-of-squares (expr)
  `(sum (* ,expr ,expr)))
```

but if you are an experienced macro writer, you will realize that this code will evaluate *expr* twice, which is probably a bad idea. A better version would use a temporary:

```
(defmacro sum-of-squares (expr)
  (let ((temp (gensym)))
    `(let ((,temp ,expr))
       (sum (* ,temp ,temp)))))
```

Although this may seem complex, it is just the sort of thing you'd have to go through to write any macro, which illustrates the point of this section: if you can write macros, you can extend `iterate`.

Our macros don't use `iterate`'s keyword-argument syntax. We could just use keywords with `defmacro`, but we would still not be using `iterate`'s clause indexing mechanism. Unlike Lisp, which uses just the first symbol of a form to determine what function to call, `iterate` individuates clauses by the list of required keywords. For instance, `for`...`in` and `for`...`in-vector` are different clauses implemented by distinct Lisp functions.

To buy into this indexing scheme, as well as the keyword-argument syntax, use `defmacro-clause`:

`defmacro-clause` *arglist* `&body` *body* [*Macro*]

> Defines a new `iterate` clause. *arglist* is a list of symbols which are alternating keywords and arguments. `&optional` may be used, and the list may be terminated by `&sequence`. *body* is an ordinary macro body, as with `defmacro`. If the first form of *body* is a string, it is considered a documentation string and will be shown by `display-iterate-clauses`. `defmacro-clause` will signal an error if defining the clause would result in an ambiguity. E.g. you cannot define the clause `for`...`from` because there would be no way to distinguish it from a use of the `for` clause with optional keyword `from`.

Here is `multiply` using `defmacro-clause`. The keywords are capitalized for readability.

```
(defmacro-clause (MULTIPLY expr &optional INTO var)
  `(reducing ,expr by #'* into ,var initial-value 1))
```

You don't have to worry about the case when `var` is not supplied; for any clause with an `into` keyword, saying `into nil` is equivalent to omitting the `into` entirely.

As another, more extended example, consider the fairly common iteration pattern that involves finding the sequence element that maximizes (or minimizes) some function. `iterate` provides this as `finding`...`maximizing`, but it's instructive to see how to write it. Here, in pseudocode, is how you might write such a loop for maximizing a function F:

```
set variable MAX-VAL to NIL;
set variable WINNER to NIL;
for each element EL in the sequence
  if MAX-VAL is NIL or F(EL) > MAX-VAL then
    set MAX-VAL to F(EL);
    set WINNER to EL;
  end if;
end for;
return WINNER.
```

Here is the macro:

```
(defmacro-clause (FINDING expr MAXIMIZING func &optional INTO var)
  (let ((max-val (gensym))
        (temp1 (gensym))
        (temp2 (gensym))
        (winner (or var iterate::*result-var*)))
    `(progn
       (with ,max-val = nil)
       (with ,winner = nil)
       (cond
        ((null ,max-val)
         (setq ,winner ,expr)
         (setq ,max-val (funcall ,func ,winner)))
        (t
         (let* ((,temp1 ,expr)
                (,temp2 (funcall ,func ,temp1)))
           (when (> ,temp2 ,max-val)
             (setq ,max-val ,temp2)
             (setq ,winner ,temp1))))))
       (finally (leave ,winner)))))
```

Note that if no `into` variable is supplied, we use `iterate::*result-var*`, which contains the internal variable into which all clauses place their results. If this variable is bound by some clause, then `iterate` will return its value automatically; otherwise, `nil` will be returned.

7.2 Writing Drivers

In principle, drivers can be implemented just as easily as other `iterate` clauses. In practice, they are a little harder to get right. As an example, consider writing a driver that iterates over all the elements of a vector, ignoring its fill-pointer. `for`...`in-vector` won't work for this, because it observes the fill-pointer. It's necessary to use `array-dimension` instead of `length` to obtain the size of the vector. Here is one approach:

```
(defmacro-clause (FOR var IN-WHOLE-VECTOR v)
  "All the elements of a vector (disregards fill-pointer)"
  (let ((vect (gensym))
        (end (gensym))
        (index (gensym)))
    `(progn
       (with ,vect = ,v)
       (with ,end = (array-dimension ,vect 0))
       (for ,index from 0 below ,end)
       (dsetq ,var (aref ,vect ,index)))))
```

Note that we immediately put `v` in a variable, in case it is an expression. Again, this is just good Lisp macrology. It also has a subtle effect on the semantics of the driver: `v` is evaluated only once, at the beginning of the loop, so changes to `v` in the loop have no effect on the driver. This is how all of `iterate`'s drivers work.

There is an important point concerning the `progn` in this code. We need the `progn`, of course, because we are returning several forms, one of which is a driver. But `iterate` drivers must occur at top-level. Is this code in error? No, because *top-level* is defined in `iterate` to include forms inside a `progn`. This is just the definition of top-level that Common Lisp uses, and for the same reason: to allow macros to return multiple forms at top-level.

While our `for`...`in-whole-vector` clause will work, it is not ideal. In particular, it does not support generating. Do do so, we need to use `for`...`next` or `for`...`do-next`. The job is simplified by the `defmacro-driver` macro.

`defmacro-driver` *arglist* `&body` *body* [*Macro*]

> Defines a driver clause in both the `for` and `generate` forms, and provides a parameter `generator?` which *body* can examine to determine how it was invoked. *arglist* is as in `defmacro-clause`, and should begin with the symbol `for`.

With `defmacro-driver`, our driver looks like this:

```
(defmacro-driver (FOR var IN-WHOLE-VECTOR v)
  "All the elements of a vector (disregards fill-pointer)"
   (let ((vect (gensym))
         (end (gensym))
         (index (gensym))
         (kwd (if generator? 'generate 'for))
     `(progn
        (with ,vect = ,v)
        (with ,end = (array-dimension ,vect 0))
        (with ,index = -1)
        (,kwd ,var next (progn (incf ,index)
                               (if (>= ,index ,end) (terminate))
                               (aref ,vect ,index))))))
```

We are still missing one thing: the `&sequence` keywords. We can get them easily enough, by writing

```
(defmacro-driver (FOR var IN-WHOLE-VECTOR v &sequence)
  ...)
```

We can now refer to parameters `from`, `to`, `by`, etc. which contain either the values for the corresponding keyword, or `nil` if the keyword was not supplied. Implementing the right code for these keywords is cumbersome but not difficult; it is left as an exercise. But before you begin, see `defclause-sequence` below for an easier way.

7.3 Extensibility Aids

This section documents assorted features that may be of use in extending `iterate`.

`iterate::*result-var*` [*Variable*]

> Holds the variable that is used to return a value as a result of the `iterate` form. You may examine this and use it in a `with` clause, but you should not change it.

`defsynonym` *syn word* [*Macro*]

> Makes *syn* a synonym for the existing `iterate` keyword *word.* Only the first word in each clause can have synonyms.

`defclause-sequence` *element-name index-name* [*Macro*]
`&key` *access-fn size-fn sequence-type*
element-type element-doc-string index-doc-string

> Provides a simple way to define sequence clauses. Generates two clauses, one for iterating over the sequence's elements, the other for iterating over its

indices. The first symbol of both clauses will have print-name `for`. *element-name* and *index-name* should be symbols. *element-name* is the second keyword of the element iterator (typically of the form `in-`*sequence-type*), and *index-name* is the second keyword of the index-iterator (typically of the form `index-of-`*sequence-type*). Either name may be `nil`, in which case the corresponding clause is not defined. If both symbols are supplied, they should be in the same package. The `for` that begins the clauses will be in this package.

access-fn is the function to be used to access elements of the sequence in the element iterator. The function should take two arguments, a sequence and an index, and return the appropriate element. *size-fn* should denote a function of one argument, a sequence, that returns its size. Both *access-fn* and *size-fn* are required for the element iterator, but only *size-fn* is needed for the index iterator.

The *sequence-type* and *element-type* keywords are used to suggest types for the variables used to hold the sequence and the sequence elements, respectively. The usual rules about `iterate`'s treatment of variable type declarations apply (see section 4).

element-doc-string and *index-doc-string* are the documentation strings, for use with `display-iterate-clauses`.

The generated element-iterator performs destructuring on the element variable.

As an example, the above `for...in-whole-vector` example could have been written:

```
(defclause-sequence IN-WHOLE-VECTOR INDEX-OF-WHOLE-VECTOR
  :access-fn 'aref
  :size-fn #'(lambda (v) (array-dimension v 0))
  :sequence-type 'vector
  :element-type t
  :element-doc-string
     "Elements of a vector, disregarding fill-pointer"
  :index-doc-string
     "Indices of vector, disregarding fill-pointer")
```

7.4 Subtleties

There are some subtleties to be aware of when writing `iterate` clauses. First, the code returned by your macros may be `nconc`'ed into a list, so you should always returned freshly consed lists, rather than constants. Second, `iterate` matches clauses by using `eq` on the first symbol and `string=` on the subsequent ones, so the package of the first symbol of a clause is relevant. All of the clauses in this manual have their first word in the `iterate` package. You can use the package system in the usual way to shadow `iterate` clauses without replacing them.

8 Obtaining `Iterate`

`iterate` currently runs on Lisp Machines, and on HP's, Sun3's and Sparcstations under Lucid. `iterate` source and binaries are available at the MIT AI Lab in the subdirectories of `/src/local/lisplib/`. The source file, `iterate.lisp`, is also available for anonymous FTP in the directory `/com/fpt/pub/` on the machine `TRIX.AI.MIT.EDU` (Internet number 128.52.32.6). If you are unable to obtain `iterate` in one of these ways, send mail to `jba@ai.mit.edu` and I will send you the source file.

`iterate` resides in the `iterate` package (nickname `iter`). Just say `(use-package "ITERATE")` to make all the necessary symbols available. If a symbol is not exported, it appears in this manual with an "`iterate::`" prefix.

Send bug reports to `bug-iterate@ai.mit.edu`. The `info-iterate` mailing list will have notices of changes and problems; to have yourself added, send mail to `info-iterate-request@ai.mit.edu`.

Acknowledgements

Richard Waters provided invaluable criticism which spurred me to improve `iterate` greatly. As early users, David Clemens, Oren Etzioni and Jeff Siskind helped ferret out many bugs.

Index

always-declare-variables 21
list-end-test 5
result-var 27
accumulate 13
adjoining 13
after-each 15
always 14
appending 13
collect 12
counting 12
declare-variables 21
defclause-sequence 27
defclause-sequence 27
defmacro-clause 24
defmacro-driver 26
defsynonym 27
display-iterate-clauses 18
dsetq 18
finally 16
finally-protected 16
finding...maximizing 14
finding...minimizing 14
finding...such-that 13
finish 15
for 4
for...= 9
for...do-next 6
for...first...then 10
for...in 4
for...in-file 6
for...in-hashtable 5
for...in-package 6
for...in-sequence 5
for...in-stream 6
for...in-string 5
for...in-vector 5
for...index-of-sequence 5
for...index-of-string 5
for...index-of-vector 5
for...initially...then 10
for...next 6
for...on 5
for...previous 8
if-first-time 15
in 17
initially 15
leave 15
maximize 12
minimize 12
multiply 11
nconcing 13
never 14
next-iteration 15
nunioning 13
reducing 12
repeat 3
sum 11
terminate 6
the 19
thereis 14
unioning 13
until 15
while 15
with 9