Unified `for` Loop

Version 7.2

Michael M. MacLeod <michaelmmacleod@gmail.com>

May 9, 2019

```
(require unified-for)
```
package: `unified-for`

This package consolidates the various §3.18.1 “Iteration and Comprehension Forms” into a single `for` macro that compiles directly to efficient named let code. It also allows identifiers to be bound via match patterns.

Warning: this package is experimental and subject to breaking changes.

```
(for maybe-accumulator (loop-clause ...) body ...+)

maybe-accumulator =
                  | accumulator
                  | (accumulator arg-form ...)

      loop-clause = [match-pattern ...+ iterator-clause]

  iterator-clause = iterator
                  | (iterator arg-form ...)
```

Iteratively binds *match-pattern*s with *iterator*s, evaluates *body*s, and collects the results with the *accumulator*. An accumulator or iterator with no subforms can be supplied without parentheses. The default accumulator is `to-void`.

All identifiers are bound via match patterns. Each pattern must successfully match, otherwise a `exn:misc:match?` exception is thrown.

Examples:

```
> (for (to-fold [evens '()]
                [odds '()])
       ([x (from-range 10)])
    (if (even? x)
```

```
        (values (cons x evens) odds)
        (values evens (cons x odds))))
'(8 6 4 2 0)
'(9 7 5 3 1)
> (for ([key value (from-hash #hash((a . 0) (b . 1) (c . 2)))])
    (displayln (~a key ": " value)))
a: 0
c: 2
b: 1
> (for (to-vector #:length 3)
       ([(cons (? symbol? _)
               (app real-part y))
         (from-list '((k1 . 1+2i) (k2 . 2+3i) (k3 . 3+4i)))])
    y)
'#(1 2 3)
```

1 Iterators

An *iterator* is a Syntax Transformer for use in the *iterator-clause* of `for`. See §3 "Extending for" on deriving new iterators.

```
(from-list lst)

  lst : list?
```

Iterates over a `list?`. Similar to `in-list`, except that `from-list` is legal only within `for`.

Example:

```
> (for ([x (from-list '(1 2 3 4 5))])
    (display x))
12345
```

```
(from-vector vect)

  vect : vector?
```

Iterates over a `vector?`. Similar to `in-vector`, except that `from-vector` is legal only within `for`.

Example:

```
> (for ([x (from-vector #(1 2 3 4 5))])
    (display x))
12345
```

```
(from-range option)

option = end-expr
       | start-expr end-expr
       | start-expr end-expr step-expr

  end-expr : real?
  start-expr : real?
  end-expr : real?
```

Iterates over a range of `real?` values from `start` (inclusive) until `end` (exclusive) by `step`. Similar to `in-range`, except that `from-range` is legal only within `for`.

If *start-expr* or *step-expr* are not provided, they are `0` and `1` respectively.

Examples:

```
> (for ([x (from-range 5)])
    (display x))
01234
> (for ([x (from-range 5 10)])
    (display x))
56789
> (for ([x (from-range 10 0 -2)])
    (display x))
108642
```

```
(from-naturals maybe-start)

maybe-start =
            | start-expr

  maybe-start : exact-nonnegative-integer?
```

Iterates forever over `natural?` numbers beginning with `start`, or `0` if `start` is not supplied. Similar to `in-naturals`, except that `from-naturals` is legal only within `for`.

Examples:

```
> (for ([index from-naturals]
        [v (from-list '(a b c d e f g))])
    (display (cons index v)))
(0 . a)(1 . b)(2 . c)(3 . d)(4 . e)(5 . f)(6 . g)
> (for ([index+1 (from-naturals 1)]
        [v (from-list '(a b c d e f g))])
    (display (cons index+1 v)))
(1 . a)(2 . b)(3 . c)(4 . d)(5 . e)(6 . f)(7 . g)
```

```
(from-hash hash-expr)

  hash-expr : hash?
```

Iterates over the keys and values of a `hash?`. Similar to `in-hash`, except that `from-hash` is legal only within `for`. Note that unlike for from `racket/base`, there must be no parentheses around the key and value *match-pattern*s.

Example:

```
> (for ([key value (from-hash #hash((a . 1) (b . 2) (c . 3)))])
    (display (cons key value)))
(a . 1)(c . 3)(b . 2)
```

2 Accumulators

An *accumulator* is a syntax transformer for use in the *maybe-accumulator* clause of `for`. See §3 “Extending for” on deriving new accumulators.

```
(to-void)
```

Returns `#<void>`. Similar to for. The result of the `for`’s *body* clause is ignored. It is the default accumulator when none is supplied to `for`.

Examples:

```
> (for to-void
        ([x (from-range 5)]
         [y (from-range 4 0 -1)])
    (define x+y (+ x y))
    (display x+y)
    x+y)
4444
> (for ([x (from-range 5)]
         [y (from-range 4 0 -1)])
    (define x+y (+ x y))
    (display x+y)
    x+y)
4444
```

```
(to-list maybe-reverse?)

maybe-reverse? =
               | #:reverse? reverse?-expr

  reverse?-expr : boolean?
```

Accumulates elements into a `list?`. Similar to `for/list`.

If `#:reverse?` is not provided, or *reverse?-expr* evaluates to `#t`, `to-list` accumulates items like `for/list`. Otherwise, `to-list` returns items in the opposite order.

The `to-list` accumulator normally collects elements in reverse order by `cons`ing them together, then applying `reverse` to the result. With `#:reverse? #f`, `to-list` does not `reverse` the result. This can give better performance.

Examples:

```
> (for to-list
        ([x (from-range 5)])
    (* x 2))
'(0 2 4 6 8)
```

```
> (for (to-list #:reverse? #f)
       ([x (from-range 5)])
    (* x 2))
'(8 6 4 2 0)
```

```
(to-vector length-option)

    length-option =
                  | expandable-option
                  | fixed-option

expandable-option = #:grow-from initial-capacity-expr
                  | #:grow-from initial-capacity-expr #:by multiplier-expr

     fixed-option = #:length length-expr
                  | #:length length-expr #:fill fill-expr

  initial-capacity-expr : exact-positive-integer?
  length-expr : exact-nonnegative-integer?
  fill-expr : any/c
  multiplier-expr : (and/c exact-integer? (>/c 1))
```

Accumulates elements into a mutable `vector?`. Similar to `for/vector`.

If *`expandable-option`* is supplied, `to-vector` will copy the existing values to a fresh mutable `vector?` each time iteration exceeds its length. The size of the new vector is calculated as `(* old-length multiplier-expr)`. The vector is trimmed to the correct size when iteration concludes.

When no arguments are supplied, `to-vector` uses the *`expandable-option`*s `#:grow-from 16 #:by 2`.

Examples:

```
> (for to-vector
       ([x (from-range 5)])
    (* x 2))
'#(0 2 4 6 8)
> (for (to-vector #:grow-from 1
                  #:by 3)
       ([x (from-range 5)])
    (* x 2))
'#(0 2 4 6 8)
```

If *`fixed-option`* is supplied, `to-vector` creates a single mutable `vector?`. Iteration is stopped as soon as the vector is completely filled. The *`length-expr`* option specifies the

size of the vector, and *fill-expr* specifies what to place in the vector if it is not completely filled by iteration. By default, *fill-expr* is 0.

Supplying a length via `#:length` *length-expr* can be more efficient than the default behavior, since the accumulator will only ever create one vector.

Examples:

```
> (for (to-vector #:length 10)
       ([x (from-range 5)])
    (* x 2))
'#(0 2 4 6 8 0 0 0 0 0)
> (for (to-vector #:length 10 #:fill #f)
       ([x (from-range 5)])
    (* x 2))
'#(0 2 4 6 8 #f #f #f #f #f)
> (for (to-vector #:length 5)
       ([x (from-range 10)])
    (display x)
    x)
01234
'#(0 1 2 3 4)
```

```
(to-fold [arg-id init-expr] ... maybe-result)

maybe-result =
             | #:result
             | result-form

  init-expr : any/c
```

Accumulates elements into any number of *arg-id*s. Similar to `for/fold`.

The *init-expr*s are evaluated and bound to *arg-id*s in the *body* forms of the `for` loop. The body of the `for` loop must evaluate to as many `values` as there are *arg-id*s. These `values` are then bound to each *arg-id* in the next iteration.

If *result-form* is supplied, it is evaluated at the end of iteration and its result returned. By default, *result-form* is `(values` *arg-id* `...)`.

Examples:

```
> (for (to-fold [sum 0]
                #:result (* 2 sum))
       ([n (from-range 10)])
    (+ sum n))
90
> (for (to-fold [real-parts '()]
                [imag-parts '()])
```

```
      ([c (from-list '(1+1i 2+5i 4+2i 9+5i))])
    (values (cons (real-part c) real-parts)
            (cons (imag-part c) imag-parts)))
'(9 4 2 1)
'(5 2 5 1)
```

3 Extending for