Unified `for` Loops

Version 7.2

Michael M. MacLeod <mmmacleo@ucsd.edu>

May 5, 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.

The unified `for` gains its functionality through §1 "Iterators" and §2 "Accumulators". It also allows identifiers to be bound with match patterns.

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

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

         loop-clause = [maybe-match-patterns iterator-clause]

maybe-match-patterns = id ...
                     | match-pattern-expr ...

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

Iteratively evaluates `body`s.

1 Iterators

An *iterator* is a syntax transformer for use in the `iterator-clause` of `for`.

```
(from-list lst)

  lst : list?
```

Iterates over a `list?`.

Example:

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

```
(from-vector vect)

  vect : vector?
```

Iterates over a `vector?`.

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`. 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.

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?`.

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`.

```
(to-void)
```

Returns `(void)`. The result of the `for`'s *body* clause is ignored. It is the default accumulator when none is provided 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 single values into a `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.

Using `#:reverse #f` can be more efficient than the default behavior. See Performance: to-list for more information.

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 growth-option

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

    growth-option = #:by multiplier-expr
                    | #:with growth-proc

  initial-capacity-expr : exact-positive-integer?
  length-expr : exact-nonnegative-integer?
  fill-expr : any/c
  multiplier-expr : (and/c exact-integer? (>=/c 2))
                (->i ([old-size exact-positive-integer?])
  growth-proc :      [new-size (old-size)
                      (and/c exact-integer? (>/c old-size))])
```

Accumulates single values into a mutable `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 determined by *`growth-option`*. If `#:by` *`multiplier-expr`* is supplied, the length of the new vector will be `(* old-length multiplier-expr)`. If `#:with` *`growth-proc`* is supplied, the length will be `(growth-proc old-length)`. The vector is trimmed to the correct size when iteration concludes.

When no options are supplied, `to-vector` uses the *`expandable-option`*s `#:grow-from 16 #:by 2`, which equivalent to how `for/vector` functions when no options are supplied.

Examples:

```
> (for to-vector
       ([x (from-range 5)])
    (* x 2))
'#(0 2 4 6 8)
> (for (to-vector #:grow-from 10
                  #:by 2)
       ([x (from-range 5)])
    (* x 2))
'#(0 2 4 6 8)
> (for (to-vector #:grow-from 10
                  #:with (λ (old-length)
                           (+ old-length 10)))
       ([x (from-range 5)])
    (* x 2))
'#(0 2 4 6 8)
```

If *fixed-option* is supplied, `to-vector` creates a single mutable `vector?` at the beginning of iteration. If iteration exceeds the length of the vector, results are silently ignored. 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. See Performance: to-vector for more information.

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)
```

3 Performance

The performance of for depends in part upon the accumulator and iterators supplied. All iterators and accumulators provided from this package perform on-par with their racket counterparts, with some including extra functionality, like to-list's #:reverse?, and to-vector's #:grow-from, which can result in improved performance when used properly.

The for syntax only expands into code that uses match if non-identifier patterns are used. Otherwise, it is expanded directly into code that uses let-values. This improves iteration speed by a small amount and reduces compiled bytecode sizes.

3.1 to-list

The to-list accumulator collects items by consing them together. Since this strategy produces a list in the opposite order of iteration, to-list reverses the result by default. If #:reverse? #f is supplied, to-list does not reverse the result, which improves performance.

3.2 to-vector

Supplying #:length *length-expr* in to-vector ensures that only one vector is ever created. This has the potential to perform faster than the default behavior of allocating a new vector when iteration exceeds the old vector's length.