Unified `for` Loop

Version 7.2

Michael M. MacLeod <michaelmmacleod@gmail.com>

May 10, 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

Creating a new iterator or accumulator involves using `define-syntax` to make a Syntax Transformer that expands into a `syntax` list. It is similar to the process of using `:do-in` to extend the traditional for loop. The `for` macro `local-expand`s each iterator and accumulator and splices their results into its own expansion.

```
    iterator = (([(outer-id ...) outer-expr] ...)
                (outer-check-expr ...)
                ([loop-id loop-expr] ...)
                pos-guard-expr
                ([(inner-id ...) inner-expr] ...)
                pre-guard-expr
                match-expr
                post-guard-expr
                (loop-arg-expr ...))

 accumulator = (([(outer-id ...) outer-expr] ...)
                (outer-check ...)
                ([loop-id loop-expr] ...)
                pos-guard-expr
                ([(inner-id ...) inner-expr] ...)
                pre-guard-expr
                (body-result-id ...)
                post-guard-expr
                (loop-arg-expr ...)
                done-expr)
```

Both accumulators and iterators expand to similar forms. The first element, `([(outer-id ...) outer-expr] ...)` specifies identifiers and expressions to be bound via `let*-values` outside the loop. This is useful when the iterator or accumulator needs to evaluate an expression only once. For example, `to-vector` with the `#:fill` option creates its `vector?` here.

The second element, `(outer-check-expr ...)`, specifies a list of expressions which are evaluated for their side effects, after `outer-id`s are bound, and before the loop begins. This is useful for checking that all sub-forms of the iterator or accumulator are of valid types. For example, `from-range` uses this space to throw an exception if its sub-forms do not evaluate to `real?` numbers.

Next is `([loop-id loop-expr] ...)`. These identifiers are bound to their expressions at the start of the loop, once all `outer-check`s have been evaluated. Later, during the iteration of the `for` form, they are bound to the result of evaluating the `loop-arg`s. For example, the `from-list` iterator binds `loop-id` to the `list?` being iterated over. The `to-fold` accumulator uses these bindings to keep track of its `arg-id`s and their bindings.

The *pos-guard-expr* form is evaluated once at the beginning of each iteration of the loop. If it produces a `#t` value, the loop continues. Otherwise, iteration ends immediately, and the accumulator's *done-expr* is returned. This form is useful for checking whether the sequence being iterated over is empty or not. For example, `from-vector` uses this space to ensure that the current index in the vector, which it bound as a *loop-arg* is less than its length. The `from-naturals` iterator expands here to `#t` its iteration is infinite.

After each *pos-guard-expr* is checked, `([(inner-id ...) inner-expr] ...)` is bound via `let*-values`. This is useful for creating bindings that differ on each iteration, and happen before the evaluation of `for`'s *body*s.

After *inner-id*s are bound, the *pre-guard-expr* is evaluated. If it produces a `#t` value, the loop continues. Otherwise, iteration ends immediately and the accumulator's *done-expr* is returned. This can be useful for ending iteration based off of a value bound to an *inner-id*.

The next form is different for iterators and accumulators. For iterators, it is *match-expr*, and it specifies what expression to match against `for`'s *match-pattern*s. For example, `from-hash`'s *match-expr* evaluates to two `values`, the current key and value of the `hash?` being iterated over. For accumulators, this form is `(body-result-id ...)`. It specifies the identifiers to bind via `let-values` to the result of `for`'s *body*s. The `to-list` accumulator supplies one identifier here, which it `cons`es onto its *loop-id* in its *loop-arg-expr*.

Both iterators and accumulators then have a *post-guard-expr*. If *post-guard-expr* evaluates to a `#t` value, the loop continues. Otherwise, iteration ends immediately and the accumulator's *done-expr* is returned. This can be useful for ending iteration based off of a value bound to a *body-result-id* in the case of accumulators, or a side effect of `for`'s *body*s, in the case of iterators.

The `(loop-arg-expr ...)` form is then evaluated, and its result is bound to each *loop-id*s on the next iteration. An iterator, like `from-vector`, uses this form to step to the next element in the sequence, usually by adding `1` to an index, or using a `cdr`-like operation. An accumulator, like `to-list`, uses this form to add an element to its collection, usually the one bound by *body-result-id*.

Each accumulator must specify one more form, *done-expr*, which is evaluated and returned whenever any *pos-guard-expr*, *pre-guard-expr*, or *post-gurd-expr* returns a `#f` value. For example, `to-list` with `#:reverse? #t` uses this space to `reverse` the accumulated list bound to its *loop-id*.

Here is the full expansion of a `for` form, with one accumulator bound to `a`, and any number of iterators bound to `(i ...)`.

```
(let*-values ([(a.outer-id ...) a.outer-expr] ...
              [(i.outer-id ...) i.outer-expr] ... ...)
  a.outer-check-expr ...
  i.outer-check-expr ... ...
```

```
(let loop ([a.loop-id a.loop-expr] ...
           [i.loop-id i.loop-expr] ... ...)
  (if (and a.pos-guard-expr i.pos-guard-expr ...)
      (let*-values ([(a.inner-id ...) a.inner-expr] ...
                    [(i.inner-id ...) i.inner-expr] ... ...)
        (if (and a.pre-guard-expr i.pre-guard-expr ...)
            (let-values ([(a.body-result-id ...)
                          (match-let-values
                              ([(pattern ...) i.match-expr] ...)
                           body ...)])
              (if (and a.post-guard-expr i.post-guard-expr ...)
                  (loop a.loop-arg-expr ... i.loop-arg-
expr ... ...)
                  a.done-expr))
            a.done-expr))
      a.done-expr)))
```

Examples:

```
> (require (for-syntax racket/base syntax/parse)
           (prefix-in u: unified-for))
> (define-syntax (from-vector stx)
    (syntax-parse stx
      [(_ v:expr)
       #`(([(vect) v]
           [(len)
            #,(syntax/loc #'v
                (vector-length vect))])
          ()
          ([pos 0])
          (< pos len)
          ()
          #t
          (vector-ref vect pos)
          #t
          ((add1 pos)))]))
> (u:for ([x (from-vector #(0 1 2 3 4 5))])
    (display x))
012345
> (u:for ([x (from-vector 'not-a-vector)])
    (display x))
vector-length: contract violation
  expected: vector?
  given: 'not-a-vector
> (define-syntax (to-fold stx)
    (syntax-parse stx
```

```
      #:track-literals
      [(_ [arg:id val:expr] ...+)
       #'(to-fold [arg val] ... #:result (values arg ...))]
      [(_ [arg:id val:expr] ...+ #:result result:expr)
       (with-syntax ([(last-body ...) (generate-
temporaries #'([arg val] ...))])
         #'(()
            ()
            ([arg val] ...)
            #t
            ()
            #t
            (last-body ...)
            #t
            (last-body ...)
            result))]))
> (u:for (to-fold [factorial 1])
         ([x (u:from-range 1 10)])
    (* factorial x))
362880
> (define-syntax (to-void stx)
    (syntax-parse stx
      [(_)
       #'(() () () #t () #t (_) #t () (void))]))
> (u:for to-void
         ([x (u:from-range 5)])
    (display x))
01234
```