NAME

List::Compare − Compare elements of two or more lists

VERSION

This document refers to version 0.53 of List::Compare. This version was released June 07 2015.

SYNOPSIS

The bare essentials:

```
@Llist = qw(abel abel baker camera delta edward fargo golfer);
@Rlist = qw(baker camera delta delta edward fargo golfer hilton);

$lc = List::Compare->new(\@Llist, \@Rlist);

@intersection = $lc->get_intersection;
@union = $lc->get_union;
```

... and so forth.

DISCUSSION: Modes and Methods

Regular Case: Compare Two Lists

- Constructor: `new()`

  Create a List::Compare object. Put the two lists into arrays (named or anonymous) and pass references to the arrays to the constructor.

  ```
  @Llist = qw(abel abel baker camera delta edward fargo golfer);
  @Rlist = qw(baker camera delta delta edward fargo golfer hilton);

  $lc = List::Compare->new(\@Llist, \@Rlist);
  ```

  By default, List::Compare's methods return lists which are sorted using Perl's default `sort` mode: ASCII-betical sorting. Should you not need to have these lists sorted, you may achieve a speed boost by constructing the List::Compare object with the unsorted option:

  ```
  $lc = List::Compare->new('-u', \@Llist, \@Rlist);
  ```

  or

  ```
  $lc = List::Compare->new('--unsorted', \@Llist, \@Rlist);
  ```

- Alternative Constructor

  If you prefer a more explicit delineation of the types of arguments passed to a function, you may use this 'single hashref' kind of constructor to build a List::Compare object:

  ```
  $lc = List::Compare->new( { lists => [\@Llist, \@Rlist] } );
  ```

  or

  ```
  $lc = List::Compare->new( {
      lists    => [\@Llist, \@Rlist],
      unsorted => 1,
  } );
  ```

- `get_intersection()`

  Get those items which appear at least once in both lists (their intersection).

  ```
  @intersection = $lc->get_intersection;
  ```

- `get_union()`

  Get those items which appear at least once in either list (their union).

```
@union = $lc->get_union;
```

- `get_unique()`

  Get those items which appear (at least once) only in the first list.

  ```
  @Lonly = $lc->get_unique;
  @Lonly = $lc->get_Lonly;    # alias
  ```

- `get_complement()`

  Get those items which appear (at least once) only in the second list.

  ```
  @Ronly = $lc->get_complement;
  @Ronly = $lc->get_Ronly;            # alias
  ```

- `get_symmetric_difference()`

  Get those items which appear at least once in either the first or the second list, but not both.

  ```
  @LorRonly = $lc->get_symmetric_difference;
  @LorRonly = $lc->get_symdiff;       # alias
  @LorRonly = $lc->get_LorRonly;      # alias
  ```

- `get_bag()`

  Make a bag of all those items in both lists. The bag differs from the union of the two lists in that it holds as many copies of individual elements as appear in the original lists.

  ```
  @bag = $lc->get_bag;
  ```

- Return references rather than lists

  An alternative approach to the above methods: If you do not immediately require an array as the return value of the method call, but simply need a *reference* to an (anonymous) array, use one of the following parallel methods:

  ```
  $intersection_ref = $lc->get_intersection_ref;
  $union_ref        = $lc->get_union_ref;
  $Lonly_ref        = $lc->get_unique_ref;
  $Lonly_ref        = $lc->get_Lonly_ref;                # alias
  $Ronly_ref        = $lc->get_complement_ref;
  $Ronly_ref        = $lc->get_Ronly_ref;                # alias
  $LorRonly_ref     = $lc->get_symmetric_difference_ref;
  $LorRonly_ref     = $lc->get_symdiff_ref;              # alias
  $LorRonly_ref     = $lc->get_LorRonly_ref;             # alias
  $bag_ref          = $lc->get_bag_ref;
  ```

- `is_LsubsetR()`

  Return a true value if the first argument passed to the constructor ('L' for 'left') is a subset of the second argument passed to the constructor ('R' for 'right').

  ```
  $LR = $lc->is_LsubsetR;
  ```

  Return a true value if R is a subset of L.

  ```
  $RL = $lc->is_RsubsetL;
  ```

- `is_LequivalentR()`

  Return a true value if the two lists passed to the constructor are equivalent, *i.e.* if every element in the left-hand list ('L') appears at least once in the right-hand list ('R') and *vice versa.*

  ```
  $eqv = $lc->is_LequivalentR;
  $eqv = $lc->is_LeqvlntR;            # alias
  ```

- `is_LdisjointR()`

  Return a true value if the two lists passed to the constructor are disjoint, *i.e.* if the two lists have zero elements in common (or, what is the same thing, if their intersection is an empty set).

  ```
  $disj = $lc->is_LdisjointR;
  ```

- `print_subset_chart()`

  Pretty-print a chart showing whether one list is a subset of the other.

  ```
  $lc->print_subset_chart;
  ```

- `print_equivalence_chart()`

  Pretty-print a chart showing whether the two lists are equivalent (same elements found at least once in both).

  ```
  $lc->print_equivalence_chart;
  ```

- `is_member_which()`

  Determine in *which* (if any) of the lists passed to the constructor a given string can be found. In list context, return a list of those indices in the constructor's argument list corresponding to lists holding the string being tested.

  ```
  @memb_arr = $lc->is_member_which('abel');
  ```

  In the example above, `@memb_arr` will be:

  ```
  ( 0 )
  ```

  because `'abel'` is found only in `@Al` which holds position `0` in the list of arguments passed to `new()`.

  In scalar context, the return value is the number of lists passed to the constructor in which a given string is found.

  As with other List::Compare methods which return a list, you may wish the above method returned a (scalar) reference to an array holding the list:

  ```
  $memb_arr_ref = $lc->is_member_which_ref('baker');
  ```

  In the example above, `$memb_arr_ref` will be:

  ```
  [ 0, 1 ]
  ```

  because `'baker'` is found in `@Llist` and `@Rlist`, which hold positions `0` and `1`, respectively, in the list of arguments passed to `new()`.

  Note: methods `is_member_which()` and `is_member_which_ref` test only one string at a time and hence take only one argument. To test more than one string at a time see the next method, `are_members_which()`.

- `are_members_which()`

  Determine in *which* (if any) of the lists passed to the constructor one or more given strings can be found. The strings to be tested are placed in an array (named or anonymous); a reference to that array is passed to the method.

  ```
  $memb_hash_ref =
      $lc->are_members_which([ qw| abel baker fargo hilton zebra | ]);
  ```

  Note: In versions of List::Compare prior to 0.25 (April 2004), the strings to be tested could be passed as a flat list. This is no longer possible; the argument must now be a reference to an array.

  The return value is a reference to a hash of arrays. The key for each element in this hash is the string being tested. Each element's value is a reference to an anonymous array whose elements are those indices in the constructor's argument list corresponding to lists holding the strings being tested. In the examples above, `$memb_hash_ref` will be:

```
{
     abel      => [ 0     ],
     baker     => [ 0, 1 ],
     fargo     => [ 0, 1 ],
     hilton    => [    1 ],
     zebra     => [       ],
};
```

Note: `are_members_which()` can take more than one argument; `is_member_which()` and `is_member_which_ref()` each take only one argument. Unlike those two methods, `are_members_which()` returns a hash reference.

- `is_member_any()`

  Determine whether a given string can be found in *any* of the lists passed as arguments to the constructor. Return 1 if a specified string can be found in any of the lists and 0 if not.

  ```
  $found = $lc->is_member_any('abel');
  ```

  In the example above, `$found` will be `1` because `'abel'` is found in one or more of the lists passed as arguments to `new()`.

- `are_members_any()`

  Determine whether a specified string or strings can be found in *any* of the lists passed as arguments to the constructor. The strings to be tested are placed in an array (named or anonymous); a reference to that array is passed to `are_members_any`.

  ```
  $memb_hash_ref = $lc->are_members_any([ qw| abel baker fargo hilton zebra
  ```

  Note: In versions of List::Compare prior to 0.25 (April 2004), the strings to be tested could be passed as a flat list. This is no longer possible; the argument must now be a reference to an array.

  The return value is a reference to a hash where an element's key is the string being tested and the element's value is 1 if the string can be found in *any* of the lists and 0 if not. In the examples above, `$memb_hash_ref` will be:

  ```
  {
       abel      => 1,
       baker     => 1,
       fargo     => 1,
       hilton    => 1,
       zebra     => 0,
  };
  ```

  `zebra`'s value is `0` because `zebra` is not found in either of the lists passed as arguments to `new()`.

- `get_version()`

  Return current List::Compare version number.

  ```
  $vers = $lc->get_version;
  ```

Accelerated Case: When User Only Wants a Single Comparison

- Constructor `new()`

  If you are certain that you will only want the results of a *single* comparison, computation may be accelerated by passing `'-a'` or `'--accelerated` as the first argument to the constructor.

```
@Llist = qw(abel abel baker camera delta edward fargo golfer);
@Rlist = qw(baker camera delta delta edward fargo golfer hilton);

$lca = List::Compare->new('-a', \@Llist, \@Rlist);
```

or

```
$lca = List::Compare->new('--accelerated', \@Llist, \@Rlist);
```

As with List::Compare's Regular case, should you not need to have a sorted list returned by an accelerated List::Compare method, you may achieve a speed boost by constructing the accelerated List::Compare object with the unsorted option:

```
$lca = List::Compare->new('-u', '-a', \@Llist, \@Rlist);
```

or

```
$lca = List::Compare->new('--unsorted', '--accelerated', \@Llist, \@Rlist)
```

- Alternative Constructor

  You may use the 'single hashref' constructor format to build a List::Compare object calling for the Accelerated mode:

```
$lca = List::Compare->new( {
    lists    => [\@Llist, \@Rlist],
    accelerated => 1,
} );
```

or

```
$lca = List::Compare->new( {
    lists    => [\@Llist, \@Rlist],
    accelerated => 1,
    unsorted => 1,
} );
```

- Methods

  All the comparison methods available in the Regular case are available to you in the Accelerated case as well.

```
@intersection     = $lca->get_intersection;
@union            = $lca->get_union;
@Lonly            = $lca->get_unique;
@Ronly            = $lca->get_complement;
@LorRonly         = $lca->get_symmetric_difference;
@bag              = $lca->get_bag;
$intersection_ref = $lca->get_intersection_ref;
$union_ref        = $lca->get_union_ref;
$Lonly_ref        = $lca->get_unique_ref;
$Ronly_ref        = $lca->get_complement_ref;
$LorRonly_ref     = $lca->get_symmetric_difference_ref;
$bag_ref          = $lca->get_bag_ref;
$LR               = $lca->is_LsubsetR;
$RL               = $lca->is_RsubsetL;
$eqv              = $lca->is_LequivalentR;
$disj             = $lca->is_LdisjointR;
                    $lca->print_subset_chart;
                    $lca->print_equivalence_chart;
@memb_arr         = $lca->is_member_which('abel');
$memb_arr_ref     = $lca->is_member_which_ref('baker');
```

```
    $memb_hash_ref     = $lca->are_members_which(
                               [ qw| abel baker fargo hilton zebra | ]);
    $found             = $lca->is_member_any('abel');
    $memb_hash_ref     = $lca->are_members_any(
                               [ qw| abel baker fargo hilton zebra | ]);
    $vers              = $lca->get_version;
```

All the aliases for methods available in the Regular case are available to you in the Accelerated case as well.

Multiple Case: Compare Three or More Lists

- Constructor `new()`

  Create a List::Compare object. Put each list into an array and pass references to the arrays to the constructor.

  ```
      @Al     = qw(abel abel baker camera delta edward fargo golfer);
      @Bob    = qw(baker camera delta delta edward fargo golfer hilton);
      @Carmen = qw(fargo golfer hilton icon icon jerky kappa);
      @Don    = qw(fargo icon jerky);
      @Ed     = qw(fargo icon icon jerky);

      $lcm = List::Compare->new(\@Al, \@Bob, \@Carmen, \@Don, \@Ed);
  ```

  As with List::Compare's Regular case, should you not need to have a sorted list returned by a List::Compare method, you may achieve a speed boost by constructing the object with the unsorted option:

  ```
      $lcm = List::Compare->new('-u', \@Al, \@Bob, \@Carmen, \@Don, \@Ed);
  ```

  or

  ```
      $lcm = List::Compare->new('--unsorted', \@Al, \@Bob, \@Carmen, \@Don, \@Ed
  ```

- Alternative Constructor

  You may use the 'single hashref' constructor format to build a List::Compare object to process three or more lists at once:

  ```
      $lcm = List::Compare->new( {
          lists    => [\@Al, \@Bob, \@Carmen, \@Don, \@Ed],
      } );
  ```

  or

  ```
      $lcm = List::Compare->new( {
          lists    => [\@Al, \@Bob, \@Carmen, \@Don, \@Ed],
          unsorted => 1,
      } );
  ```

- Multiple Mode Methods Analogous to Regular and Accelerated Mode Methods

  Each List::Compare method available in the Regular and Accelerated cases has an analogue in the Multiple case. However, the results produced usually require more careful specification.

  Note: Certain of the following methods available in List::Compare's Multiple mode take optional numerical arguments where those numbers represent the index position of a particular list in the list of arguments passed to the constructor. To specify this index position correctly,

  - start the count at `0` (as is customary with Perl array indices); and
  - do *not* count any unsorted option (`'-u'` or `'--unsorted'`) preceding the array references in the constructor's own argument list.

Example:

```
$lcmex = List::Compare->new('--unsorted', \@alpha, \@beta, \@gamma);
```

For the purpose of supplying a numerical argument to a method which optionally takes such an argument, `'--unsorted'` is skipped, `@alpha` is `0`, `@beta` is `1`, and so forth.

- `get_intersection()`

  Get those items found in *each* of the lists passed to the constructor (their intersection):

  ```
  @intersection = $lcm->get_intersection;
  ```

- `get_union()`

  Get those items found in *any* of the lists passed to the constructor (their union):

  ```
  @union = $lcm->get_union;
  ```

- `get_unique()`

  To get those items which appear only in *one particular list,* provide `get_unique()` with that list's index position in the list of arguments passed to the constructor (not counting any `'-u'` or `'--unsorted'` option).

  Example: `@Carmen` has index position `2` in the constructor's `@_`. To get elements unique to `@Carmen`:

  ```
  @Lonly = $lcm->get_unique(2);
  ```

  If no index position is passed to `get_unique()` it will default to 0 and report items unique to the first list passed to the constructor.

- `get_complement()`

  To get those items which appear in any list *other than one particular list,* provide `get_complement()` with that list's index position in the list of arguments passed to the constructor (not counting any `'-u'` or `'--unsorted'` option).

  Example: `@Don` has index position `3` in the constructor's `@_`. To get elements not found in `@Don`:

  ```
  @Ronly = $lcm->get_complement(3);
  ```

  If no index position is passed to `get_complement()` it will default to 0 and report items found in any list other than the first list passed to the constructor.

- `get_symmetric_difference()`

  Get those items each of which appears in *only one* of the lists passed to the constructor (their symmetric_difference);

  ```
  @LorRonly = $lcm->get_symmetric_difference;
  ```

- `get_bag()`

  Make a bag of all items found in any list. The bag differs from the lists' union in that it holds as many copies of individual elements as appear in the original lists.

  ```
  @bag = $lcm->get_bag;
  ```

- Return reference instead of list

  An alternative approach to the above methods: If you do not immediately require an array as the return value of the method call, but simply need a *reference* to an array, use one of the following parallel methods:

```
    $intersection_ref = $lcm->get_intersection_ref;
    $union_ref        = $lcm->get_union_ref;
    $Lonly_ref        = $lcm->get_unique_ref(2);
    $Ronly_ref        = $lcm->get_complement_ref(3);
    $LorRonly_ref     = $lcm->get_symmetric_difference_ref;
    $bag_ref          = $lcm->get_bag_ref;
```

- `is_LsubsetR()`

  To determine whether one particular list is a subset of another list passed to the constructor, provide `is_LsubsetR()` with the index position of the presumed subset (ignoring any unsorted option), followed by the index position of the presumed superset.

  Example: To determine whether `@Ed` is a subset of `@Carmen`, call:

  ```
      $LR = $lcm->is_LsubsetR(4,2);
  ```

  A true value (`1`) is returned if the left-hand list is a subset of the right-hand list; a false value (`0`) is returned otherwise.

  If no arguments are passed, `is_LsubsetR()` defaults to `(0,1)` and compares the first two lists passed to the constructor.

- `is_LequivalentR()`

  To determine whether any two particular lists are equivalent to each other, provide `is_LequivalentR` with their index positions in the list of arguments passed to the constructor (ignoring any unsorted option).

  Example: To determine whether `@Don` and `@Ed` are equivalent, call:

  ```
      $eqv = $lcm->is_LequivalentR(3,4);
  ```

  A true value (`1`) is returned if the lists are equivalent; a false value (`0`) otherwise.

  If no arguments are passed, `is_LequivalentR` defaults to `(0,1)` and compares the first two lists passed to the constructor.

- `is_LdisjointR()`

  To determine whether any two particular lists are disjoint from each other (*i.e.,* have no members in common), provide `is_LdisjointR` with their index positions in the list of arguments passed to the constructor (ignoring any unsorted option).

  Example: To determine whether `@Don` and `@Ed` are disjoint, call:

  ```
      $disj = $lcm->is_LdisjointR(3,4);
  ```

  A true value (`1`) is returned if the lists are equivalent; a false value (`0`) otherwise.

  If no arguments are passed, `is_LdisjointR` defaults to `(0,1)` and compares the first two lists passed to the constructor.

- `print_subset_chart()`

  Pretty-print a chart showing the subset relationships among the various source lists:

  ```
      $lcm->print_subset_chart;
  ```

- `print_equivalence_chart()`

  Pretty-print a chart showing the equivalence relationships among the various source lists:

  ```
      $lcm->print_equivalence_chart;
  ```

- `is_member_which()`

  Determine in *which* (if any) of the lists passed to the constructor a given string can be found. In list context, return a list of those indices in the constructor's argument list (ignoring any unsorted option) corresponding to i lists holding the string being tested.

```
@memb_arr = $lcm->is_member_which('abel');
```

In the example above, `@memb_arr` will be:

```
( 0 )
```

because `'abel'` is found only in `@Al` which holds position `0` in the list of arguments passed to `new()`.

- `is_member_which_ref()`

  As with other List::Compare methods which return a list, you may wish the above method returned a (scalar) reference to an array holding the list:

  ```
  $memb_arr_ref = $lcm->is_member_which_ref('jerky');
  ```

  In the example above, `$memb_arr_ref` will be:

  ```
  [ 3, 4 ]
  ```

  because `'jerky'` is found in `@Don` and `@Ed`, which hold positions `3` and `4`, respectively, in the list of arguments passed to `new()`.

  Note: methods `is_member_which()` and `is_member_which_ref` test only one string at a time and hence take only one argument. To test more than one string at a time see the next method, `are_members_which()`.

- `are_members_which()`

  Determine in `which` (if any) of the lists passed to the constructor one or more given strings can be found. The strings to be tested are placed in an anonymous array, a reference to which is passed to the method.

  ```
  $memb_hash_ref =
      $lcm->are_members_which([ qw| abel baker fargo hilton zebra | ]);
  ```

  Note: In versions of List::Compare prior to 0.25 (April 2004), the strings to be tested could be passed as a flat list. This is no longer possible; the argument must now be a reference to an anonymous array.

  The return value is a reference to a hash of arrays. The key for each element in this hash is the string being tested. Each element's value is a reference to an anonymous array whose elements are those indices in the constructor's argument list corresponding to lists holding the strings being tested.

  In the two examples above, `$memb_hash_ref` will be:

  ```
  {
       abel     => [ 0             ],
       baker    => [ 0, 1          ],
       fargo    => [ 0, 1, 2, 3, 4 ],
       hilton   => [    1, 2       ],
       zebra    => [               ],
  };
  ```

  Note: `are_members_which()` can take more than one argument; `is_member_which()` and `is_member_which_ref()` each take only one argument. `are_members_which()` returns a hash reference; the other methods return either a list or a reference to an array holding that list, depending on context.

- `is_member_any()`

Determine whether a given string can be found in *any* of the lists passed as arguments to the constructor.

```
$found = $lcm->is_member_any('abel');
```

Return `1` if a specified string can be found in *any* of the lists and `0` if not.

In the example above, `$found` will be `1` because `'abel'` is found in one or more of the lists passed as arguments to `new()`.

- `are_members_any()`

  Determine whether a specified string or strings can be found in *any* of the lists passed as arguments to the constructor. The strings to be tested are placed in an array (anonymous or named), a reference to which is passed to the method.

  ```
  $memb_hash_ref = $lcm->are_members_any([ qw| abel baker fargo hilton ze
  ```

  Note: In versions of List::Compare prior to 0.25 (April 2004), the strings to be tested could be passed as a flat list. This is no longer possible; the argument must now be a reference to an anonymous array.

  The return value is a reference to a hash where an element's key is the string being tested and the element's value is 1 if the string can be found in `any` of the lists and 0 if not. In the two examples above, `$memb_hash_ref` will be:

  ```
  {
       abel     => 1,
       baker    => 1,
       fargo    => 1,
       hilton   => 1,
       zebra    => 0,
  };
  ```

  `zebra`'s value will be `0` because `zebra` is not found in any of the lists passed as arguments to `new()`.

- `get_version()`

  Return current List::Compare version number:

  ```
  $vers = $lcm->get_version;
  ```

- Multiple Mode Methods Not Analogous to Regular and Accelerated Mode Methods
  - `get_nonintersection()`

    Get those items found in *any* of the lists passed to the constructor which do *not* appear in *all* of the lists (*i.e.,* all items except those found in the intersection of the lists):

    ```
    @nonintersection = $lcm->get_nonintersection;
    ```

  - `get_shared()`

    Get those items which appear in more than one of the lists passed to the constructor (*i.e.,* all items except those found in their symmetric difference);

    ```
    @shared = $lcm->get_shared;
    ```

  - `get_nonintersection_ref()`

    If you only need a reference to an array as a return value rather than a full array, use the following alternative methods:

    ```
    $nonintersection_ref = $lcm->get_nonintersection_ref;
    $shared_ref = $lcm->get_shared_ref;
    ```

- `get_unique_all()`

  Get a reference to an array of array references where each of the interior arrays holds the list of those items *unique* to the list passed to the constructor with the same index position.

  ```
  $unique_all_ref = $lcm->get_unique_all();
  ```

  In the example above, `$unique_all_ref` will hold:

  ```
  [
      [ qw| abel | ],
      [ ],
      [ qw| jerky | ],
      [ ],
      [ ],
  ]
  ```

- `get_complement_all()`

  Get a reference to an array of array references where each of the interior arrays holds the list of those items in the *complement* to the list passed to the constructor with the same index position.

  ```
  $complement_all_ref = $lcm->get_complement_all();
  ```

  In the example above, `$complement_all_ref` will hold:

  ```
  [
      [ qw| hilton icon jerky | ],
      [ qw| abel icon jerky | ],
      [ qw| abel baker camera delta edward | ],
      [ qw| abel baker camera delta edward jerky | ],
      [ qw| abel baker camera delta edward jerky | ],
  ]
  ```

Multiple Accelerated Case: Compare Three or More Lists but Request Only a Single Comparison among the Lists

- Constructor `new()`

  If you are certain that you will only want the results of a single comparison among three or more lists, computation may be accelerated by passing `'-a'` or `'--accelerated` as the first argument to the constructor.

  ```
  @Al     = qw(abel abel baker camera delta edward fargo golfer);
  @Bob    = qw(baker camera delta delta edward fargo golfer hilton);
  @Carmen = qw(fargo golfer hilton icon icon jerky kappa);
  @Don    = qw(fargo icon jerky);
  @Ed     = qw(fargo icon icon jerky);

  $lcma = List::Compare->new('-a',
              \@Al, \@Bob, \@Carmen, \@Don, \@Ed);
  ```

  As with List::Compare's other cases, should you not need to have a sorted list returned by a List::Compare method, you may achieve a speed boost by constructing the object with the unsorted option:

  ```
  $lcma = List::Compare->new('-u', '-a',
              \@Al, \@Bob, \@Carmen, \@Don, \@Ed);
  ```

  or

  ```
  $lcma = List::Compare->new('--unsorted', '--accelerated',
              \@Al, \@Bob, \@Carmen, \@Don, \@Ed);
  ```

As was the case with List::Compare's Multiple mode, do not count the unsorted option (`'-u'` or `'--unsorted'`) or the accelerated option (`'-a'` or `'--accelerated'`) when determining the index position of a particular list in the list of array references passed to the constructor.

Example:

```
$lcmaex = List::Compare->new('--unsorted', '--accelerated',
                 \@alpha, \@beta, \@gamma);
```

- Alternative Constructor

  The 'single hashref' format may be used to construct a List::Compare object which calls for accelerated processing of three or more lists at once:

  ```
  $lcmaex = List::Compare->new( {
      accelerated => 1,
      lists       => [\@alpha, \@beta, \@gamma],
  } );
  ```

  or

  ```
  $lcmaex = List::Compare->new( {
      unsorted    => 1,
      accelerated => 1,
      lists       => [\@alpha, \@beta, \@gamma],
  } );
  ```

- Methods

  For the purpose of supplying a numerical argument to a method which optionally takes such an argument, `'--unsorted'` and `'--accelerated` are skipped, `@alpha` is `0`, `@beta` is `1`, and so forth. To get a list of those items unique to `@gamma`, you would call:

  ```
  @gamma_only = $lcmaex->get_unique(2);
  ```

Passing Seen-hashes to the Constructor Instead of Arrays

- When Seen-Hashes Are Already Available to You

  Suppose that in a particular Perl program, you had to do extensive munging of data from an external source and that, once you had correctly parsed a line of data, it was easier to assign that datum to a hash than to an array. More specifically, suppose that you used each datum as the key to an element of a lookup table in the form of a *seen-hash*:

  ```
  my %Llist = (
      abel     => 2,
      baker    => 1,
      camera   => 1,
      delta    => 1,
      edward   => 1,
      fargo    => 1,
      golfer   => 1,
  );

  my %Rlist = (
      baker    => 1,
      camera   => 1,
      delta    => 2,
      edward   => 1,
      fargo    => 1,
      golfer   => 1,
      hilton   => 1,
  ```

```
    );
```

In other words, suppose it was more convenient to compute a lookup table *implying* a list than to compute that list explicitly.

Since in almost all cases List::Compare takes the elements in the arrays passed to its constructor and *internally* assigns them to elements in a seen-hash, why shouldn't you be able to pass (references to) seen-hashes *directly* to the constructor and avoid unnecessary array assignments before the constructor is called?

- Constructor `new()`

  You can now do so:

  ```
      $lcsh = List::Compare->new(\%Llist, \%Rlist);
  ```

- Methods

  All of List::Compare's output methods are supported *without further modification* when references to seen-hashes are passed to the constructor.

  ```
      @intersection         = $lcsh->get_intersection;
      @union                = $lcsh->get_union;
      @Lonly                = $lcsh->get_unique;
      @Ronly                = $lcsh->get_complement;
      @LorRonly             = $lcsh->get_symmetric_difference;
      @bag                  = $lcsh->get_bag;
      $intersection_ref     = $lcsh->get_intersection_ref;
      $union_ref            = $lcsh->get_union_ref;
      $Lonly_ref            = $lcsh->get_unique_ref;
      $Ronly_ref            = $lcsh->get_complement_ref;
      $LorRonly_ref         = $lcsh->get_symmetric_difference_ref;
      $bag_ref              = $lcsh->get_bag_ref;
      $LR                   = $lcsh->is_LsubsetR;
      $RL                   = $lcsh->is_RsubsetL;
      $eqv                  = $lcsh->is_LequivalentR;
      $disj                 = $lcsh->is_LdisjointR;
                              $lcsh->print_subset_chart;
                              $lcsh->print_equivalence_chart;
      @memb_arr             = $lsch->is_member_which('abel');
      $memb_arr_ref         = $lsch->is_member_which_ref('baker');
      $memb_hash_ref        = $lsch->are_members_which(
                                  [ qw| abel baker fargo hilton zebra | ]);
      $found                = $lsch->is_member_any('abel');
      $memb_hash_ref        = $lsch->are_members_any(
                                  [ qw| abel baker fargo hilton zebra | ]);
      $vers                 = $lcsh->get_version;
      $unique_all_ref       = $lcsh->get_unique_all();
      $complement_all_ref   = $lcsh->get_complement_all();
  ```

- Accelerated Mode and Seen-Hashes

  To accelerate processing when you want only a single comparison among two or more lists, you can pass `'-a'` or `'--accelerated` to the constructor before passing references to seen-hashes.

  ```
      $lcsha = List::Compare->new('-a', \%Llist, \%Rlist);
  ```

  To compare three or more lists simultaneously, pass three or more references to seen-hashes. Thus,

  ```
      $lcshm = List::Compare->new(\%Alpha, \%Beta, \%Gamma);
  ```

  will generate meaningful comparisons of three or more lists simultaneously.

- Unsorted Results and Seen-Hashes

  If you do not need sorted lists returned, pass `'-u'` or `--unsorted` to the constructor before passing references to seen-hashes.

  ```
  $lcshu  = List::Compare->new('-u', \%Llist, \%Rlist);
  $lcshau = List::Compare->new('-u', '-a', \%Llist, \%Rlist);
  $lcshmu = List::Compare->new('--unsorted', \%Alpha, \%Beta, \%Gamma);
  ```

  As was true when we were using List::Compare's Multiple and Multiple Accelerated modes, do not count any unsorted or accelerated option when determining the array index of a particular seen-hash reference passed to the constructor.

- Alternative Constructor

  The 'single hashref' form of constructor is also available to build List::Compare objects where seen-hashes are used as arguments:

  ```
  $lcshu  = List::Compare->new( {
      unsorted => 1,
      lists    => [\%Llist, \%Rlist],
  } );

  $lcshau = List::Compare->new( {
      unsorted    => 1,
      accelerated => 1,
      lists       => [\%Llist, \%Rlist],
  } );

  $lcshmu = List::Compare->new( {
      unsorted => 1,
      lists    => [\%Alpha, \%Beta, \%Gamma],
  } );
  ```

DISCUSSION: Principles

General Comments

List::Compare is an object-oriented implementation of very common Perl code (see "History, References and Development" below) used to determine interesting relationships between two or more lists at a time. A List::Compare object is created and automatically computes the values needed to supply List::Compare methods with appropriate results. In the current implementation List::Compare methods will return new lists containing the items found in any designated list alone (unique), any list other than a designated list (complement), the intersection and union of all lists and so forth. List::Compare also has (a) methods to return Boolean values indicating whether one list is a subset of another and whether any two lists are equivalent to each other (b) methods to pretty-print very simple charts displaying the subset and equivalence relationships among lists.

Except for List::Compare's `get_bag()` method, **multiple instances of an element in a given list count only once with respect to computing the intersection, union, etc. of the two lists.** In particular, List::Compare considers two lists as equivalent if each element of the first list can be found in the second list and *vice versa*. 'Equivalence' in this usage takes no note of the frequency with which elements occur in either list or their order within the lists. List::Compare asks the question: *Did I see this item in this list at all?* Only when you use `List::Compare::get_bag()` to compute a bag holding the two lists do you ask the question: How many times did this item occur in this list?

List::Compare Modes

In its current implementation List::Compare has four modes of operation.

- Regular Mode

  List::Compare's Regular mode is based on List::Compare v0.11 — the first version of List::Compare released to CPAN (June 2002). It compares only two lists at a time. Internally, its initializer does all computations needed to report any desired comparison and its constructor stores the results of these computations. Its public methods merely report these results.

This approach has the advantage that if you need to examine more than one form of comparison between two lists (*e.g.,* the union, intersection and symmetric difference of two lists), the comparisons are pre-calculated. This approach is efficient because certain types of comparison presuppose that other types have already been calculated. For example, to calculate the symmetric difference of two lists, one must first determine the items unique to each of the two lists.

- Accelerated Mode

  The current implementation of List::Compare offers you the option of getting even faster results *provided* that you only need the result from a *single* form of comparison between two lists. (*e.g.,* only the union — nothing else). In the Accelerated mode, List::Compare's initializer does no computation and its constructor stores only references to the two source lists. All computation needed to report results is deferred to the method calls.

  The user selects this approach by passing the option flag `'-a'` to the constructor before passing references to the two source lists. List::Compare notes the option flag and silently switches into Accelerated mode. From the perspective of the user, there is no further difference in the code or in the results.

  Benchmarking suggests that List::Compare's Accelerated mode (a) is faster than its Regular mode when only one comparison is requested; (b) is about as fast as Regular mode when two comparisons are requested; and (c) becomes considerably slower than Regular mode as each additional comparison above two is requested.

- Multiple Mode

  List::Compare now offers the possibility of comparing three or more lists at a time. Simply store the extra lists in arrays and pass references to those arrays to the constructor. List::Compare detects that more than two lists have been passed to the constructor and silently switches into Multiple mode.

  As described in the Synopsis above, comparing more than two lists at a time offers you a wider, more complex palette of comparison methods. Individual items may appear in just one source list, in all the source lists, or in some number of lists between one and all. The meaning of 'union', 'intersection' and 'symmetric difference' is conceptually unchanged when you move to multiple lists because these are properties of all the lists considered together. In contrast, the meaning of 'unique', 'complement', 'subset' and 'equivalent' changes because these are properties of one list compared with another or with all the other lists combined.

  List::Compare takes this complexity into account by allowing you to pass arguments to the public methods requesting results with respect to a specific list (for `get_unique()` and `get_complement()`) or a specific pair of lists (for `is_LsubsetR()` and `is_LequivalentR()`).

  List::Compare further takes this complexity into account by offering the new methods `get_shared()` and `get_nonintersection()` described in the Synopsis above.

- Multiple Accelerated Mode

  Beginning with version 0.25, introduced in April 2004, List::Compare offers the possibility of accelerated computation of a single comparison among three or more lists at a time. Simply store the extra lists in arrays and pass references to those arrays to the constructor preceded by the `'-a'` argument as was done with the simple (two lists only) accelerated mode. List::Compare detects that more than two lists have been passed to the constructor and silently switches into Multiple Accelerated mode.

- Unsorted Option

  When List::Compare is used to return lists representing various comparisons of two or more lists (*e.g.,* the lists' union or intersection), the lists returned are, by default, sorted using Perl's default `sort` mode: ASCII-betical sorting. Sorting produces results which are more easily human-readable but may entail a performance cost.

Should you not need sorted results, you can avoid the potential performance cost by calling List::Compare's constructor using the unsorted option. This is done by calling `'-u'` or `'--unsorted'` as the first argument passed to the constructor, *i.e.*, as an argument called before any references to lists are passed to the constructor.

Note that if are calling List::Compare in the Accelerated or Multiple Accelerated mode *and* wish to have the lists returned in unsorted order, you *first* pass the argument for the unsorted option (`'-u'` or `'--unsorted'`) and *then* pass the argument for the Accelerated mode (`'-a'` or `'--accelerated'`).

Miscellaneous Methods

It would not really be appropriate to call `get_shared()` and `get_nonintersection()` in Regular or Accelerated mode since they are conceptually based on the notion of comparing more than two lists at a time. However, there is always the possibility that a user may be comparing only two lists (accelerated or not) and may accidentally call one of those two methods. To prevent fatal run-time errors and to caution you to use a more appropriate method, these two methods are defined for Regular and Accelerated modes so as to return suitable results but also generate a carp message that advise you to re-code.

Similarly, the method `is_RsubsetL()` is appropriate for the Regular and Accelerated modes but is not really appropriate for Multiple mode. As a defensive maneuver, it has been defined for Multiple mode so as to return suitable results but also to generate a carp message that advises you to re-code.

In List::Compare v0.11 and earlier, the author provided aliases for various methods based on the supposition that the source lists would be referred to as 'A' and 'B'. Now that you can compare more than two lists at a time, the author feels that it would be more appropriate to refer to the elements of two-argument lists as the left-hand and right-hand elements. Hence, we are discouraging the use of methods such as `get_Aonly()`, `get_Bonly()` and `get_AorBonly()` as aliases for `get_unique()`, `get_complement()` and `get_symmetric_difference()`. However, to guarantee backwards compatibility for the vast audience of Perl programmers using earlier versions of List::Compare (all 10e1 of you) these and similar methods for subset relationships are still defined.

List::Compare::SeenHash Discontinued Beginning with Version 0.26

Prior to v0.26, introduced April 11, 2004, if a user wished to pass references to seen-hashes to List::Compare's constructor rather than references to arrays, he or she had to call a different, parallel module: List::Compare::SeenHash. The code for that looked like this:

```
use List::Compare::SeenHash;

my %Llist = (
   abel     => 2,
   baker    => 1,
   camera   => 1,
   delta    => 1,
   edward   => 1,
   fargo    => 1,
   golfer   => 1,
);

my %Rlist = (
   baker    => 1,
   camera   => 1,
   delta    => 2,
   edward   => 1,
   fargo    => 1,
        golfer   => 1,
        hilton   => 1,
    );

    my $lcsh = List::Compare::SeenHash->new(\%Llist, \%Rlist);
```

List::Compare::SeenHash is deprecated beginning with version 0.26. All its functionality (and more) has been implemented in List::Compare itself, since a user can now pass *either* a series of array references *or* a series of seen-hash references to List::Compare's constructor.

To simplify future maintenance of List::Compare, List::Compare::SeenHash.pm will no longer be distributed with List::Compare, nor will the files in the test suite which tested List::Compare::SeenHash upon installation be distributed.

Should you still need List::Compare::SeenHash, use version 0.25 from CPAN, or simply edit your Perl programs which used List::Compare::SeenHash. Those scripts may be edited quickly with, for example, this editing command in Unix text editor *vi*:

```
    :1,$s/List::Compare::SeenHash/List::Compare/gc
```

A Non-Object-Oriented Interface: List::Compare::Functional

Version 0.21 of List::Compare introduced List::Compare::Functional, a functional (*i.e.*, non-object-oriented) interface to list comparison functions. List::Compare::Functional supports the same functions currently supported by List::Compare. It works similar to List::Compare's Accelerated and Multiple Accelerated modes (described above), bit it does not require use of the `'-a'` flag in the function call. List::Compare::Functional will return unsorted comparisons of two lists by passing `'-u'` or `'--unsorted'` as the first argument to the function. Please see the documentation for List::Compare::Functional to learn how to import its functions into your main package.

ASSUMPTIONS AND QUALIFICATIONS

The program was created with Perl 5.6. The use of *h2xs* to prepare the module's template installed `require 5.005_62;` at the top of the module. This has been commented out in the actual module as the code appears to be compatible with earlier versions of Perl; how earlier the author cannot say. In particular, the author would like the module to be installable on older versions of MacPerl. As is, the author has successfully installed the module on Linux, Windows 9x and Windows 2000. See <http://testers.cpan.org/show/List–Compare.html> for a list of other systems on which this version of List::Compare has been tested and installed.

HISTORY, REFERENCES AND DEVELOPMENT

The Code Itself

List::Compare is based on code presented by Tom Christiansen & Nathan Torkington in *Perl Cookbook* <http://www.oreilly.com/catalog/cookbook/> (a.k.a. the 'Ram' book), O'Reilly & Associates, 1998, Recipes 4.7 and 4.8. Similar code is presented in the Camel book: *Programming Perl*, by Larry Wall, Tom Christiansen, Jon Orwant. <http://www.oreilly.com/catalog/pperl3/>, 3rd ed, O'Reilly & Associates, 2000. The list comparison code is so basic and Perlish that I suspect it may have been written by Larry himself at the dawn of Perl time. The `get_bag()` method was inspired by Jarkko Hietaniemi's Set::Bag module and Daniel Berger's Set::Array module, both available on CPAN.

List::Compare's original objective was simply to put this code in a modular, object-oriented framework. That framework, not surprisingly, is taken mostly from Damian Conway's *Object Oriented Perl* <http://www.manning.com/Conway/index.html>, Manning Publications, 2000.

With the addition of the Accelerated, Multiple and Multiple Accelerated modes, List::Compare expands considerably in both size and capabilities. Nonetheless, Tom and Nat's *Cookbook* code still lies at its core: the use of hashes as look-up tables to record elements seen in lists. Please note: List::Compare is not concerned with any concept of 'equality' among lists which hinges upon the frequency with which, or the order in which, elements appear in the lists to be compared. If this does not meet your needs, you should look elsewhere or write your own module.

The Inspiration

I realized the usefulness of putting the list comparison code into a module while preparing an introductory level Perl course given at the New School University's Computer Instruction Center in April-May 2002. I was comparing lists left and right. When I found myself writing very similar functions in different scripts, I knew a module was lurking somewhere. I learned the truth of the mantra "Repeated Code is a Mistake" from a 2001 talk by Mark-Jason Dominus <http://perl.plover.com/> to the New York Perlmongers <http://ny.pm.org/>. See <http://www.perl.com/pub/a/2000/11/repair3.html>.

The first public presentation of this module took place at Perl Seminar New York <http://groups.yahoo.com/group/perlsemny> on May 21, 2002. Comments and suggestions were provided there and since by Glenn Maciag, Gary Benson, Josh Rabinowitz, Terrence Brannon and Dave Cross.

The placement in the installation tree of Test::ListCompareSpecial came as a result of a question answered by Michael Graham in his talk "Test::More to Test::Extreme" given at Yet Another Perl Conference::Canada in Ottawa, Ontario, on May 16, 2003.

In May-June 2003, Glenn Maciag made valuable suggestions which led to changes in method names and documentation in v0.20.

Another presentation at Perl Seminar New York in October 2003 prompted me to begin planning List::Compare::Functional.

In a November 2003 Perl Seminar New York presentation, Ben Holtzman discussed the performance costs entailed in Perl's `sort` function. This led me to ask, "Why should a user of List::Compare pay this performance cost if he or she doesn't need a human-readable list as a result (as would be the case if the list returned were used as the input into some other function)?" This led to the development of List::Compare's unsorted option.

An April 2004 offer by Kevin Carlson to write an article for *The Perl Journal* (<http://tpj.com>) led me to re-think whether a separate module (the former List::Compare::SeenHash) was truly needed when a user wanted to provide the constructor with references to seen-hashes rather than references to arrays. Since I had already adapted List::Compare::Functional to accept both kinds of arguments, I adapted List::Compare in the same manner. This meant that List::Compare::SeenHash and its related installation tests could be deprecated and deleted from the CPAN distribution.

A remark by David H. Adler at a New York Perlmongers meeting in April 2004 led me to develop the 'single hashref' alternative constructor format, introduced in version 0.29 the following month.

Presentations at two different editions of Yet Another Perl Conference (YAPC) inspired the development of List::Compare versions 0.30 and 0.31. I was selected to give a talk on List::Compare at YAPC::NA::2004 in Buffalo. This spurred me to improve certain aspects of the documentation. Version 0.31 owes its inspiration to one talk at the Buffalo YAPC and one earlier talk at YAPC::EU::2003 in Paris. In Paris I heard Paul Johnson speak on his CPAN module Devel::Cover and on coverage analysis more generally. That material was over my head at that time, but in Buffalo I heard Andy Lester discuss Devel::Cover as part of his discussion of testing and of the Phalanx project (<http://qa.perl.org/phalanx>). This time I got it, and when I returned from Buffalo I applied Devel::Cover to List::Compare and wrote additional tests to improve its subroutine and statement coverage. In addition, I added two new methods, `get_unique_all` and `get_complement_all`. In writing these two methods, I followed a model of test-driven development much more so than in earlier versions of List::Compare and my other CPAN modules. The result? List::Compare's test suite grew by over 3300 tests to nearly 23,000 tests.

At the Second New York Perl Hackathon (May 02 2015), a project was created to request performance improvements in certain List::Compare functions (<https://github.com/nyperlmongers/nyperlhackathon2015/wiki/List−Compare−Performance−Improvements>). Hackathon participant Michael Rawson submitted a pull request with changes to List::Compare::Base::_Auxiliary. After these revisions were benchmarked, a patch embodying the pull request was accepted, leading to CPAN version 0.53.

If You Like List::Compare, You'll Love ...

While preparing this module for distribution via CPAN, I had occasion to study a number of other modules already available on CPAN. Each of these modules is more sophisticated than List::Compare — which is not surprising since all that List::Compare originally aspired to do was to avoid typing Cookbook code repeatedly. Here is a brief description of the features of these modules. (**Warning:** The following discussion is only valid as of June 2002. Some of these modules may have changed since then.)

- Algorithm::Diff – Compute 'intelligent' differences between two files/lists (<http://search.cpan.org/dist/Algorithm–Diff/>)

  Algorithm::Diff is a sophisticated module originally written by Mark-Jason Dominus, later maintained by Ned Konz, now maintained by Tye McQueen. Think of the Unix `diff` utility and you're on the right track. Algorithm::Diff exports methods such as `diff`, which "computes the smallest set of additions and deletions necessary to turn the first sequence into the second, and returns a description of these changes." Algorithm::Diff is mainly concerned with the sequence of elements within two lists. It does not export functions for intersection, union, subset status, etc.

- Array::Compare – Perl extension for comparing arrays (<http://search.cpan.org/dist/Array–Compare/>)

  Array::Compare, by Dave Cross, asks whether two arrays are the same or different by doing a `join` on each string with a separator character and comparing the resulting strings. Like List::Compare, it is an object-oriented module. A sophisticated feature of Array::Compare is that it allows you to specify how 'whitespace' in an array (an element which is undefined, the empty string, or whitespace within an element) should be evaluated for purpose of determining equality or difference. It does not directly provide methods for intersection and union.

- List::Util – A selection of general-utility list subroutines (<http://search.cpan.org/dist/Scalar–List–Utils/>)

  List::Util, by Graham Barr, exports a variety of simple, useful functions for operating on one list at a time. The `min` function returns the lowest numerical value in a list; the `max` function returns the highest value; and so forth. List::Compare differs from List::Util in that it is object-oriented and that it works on two strings at a time rather than just one — but it aims to be as simple and useful as List::Util. List::Util will be included in the standard Perl distribution as of Perl 5.8.0.

  Lists::Util (<http://search.cpan.org/dist/List–MoreUtils/>), by Tassilo von Parseval, building on code by Terrence Brannon, provides methods which extend List::Util's functionality.

- Quantum::Superpositions (<http://search.cpan.org/dist/Quantum–Superpositions/>), originally by Damian Conway, now maintained by Steven Lembark is useful if, in addition to comparing lists, you need to emulate quantum supercomputing as well. Not for the eigen-challenged.

- Set::Scalar – basic set operations (<http://search.cpan.org/dist/Set–Scalar/>)

  Set::Bag – bag (multiset) class (<http://search.cpan.org/dist/Set–Bag/>)

  Both of these modules are by Jarkko Hietaniemi. Set::Scalar has methods to return the intersection, union, difference and symmetric difference of two sets, as well as methods to return items unique to a first set and complementary to it in a second set. It has methods for reporting considerably more variants on subset status than does List::Compare. However, benchmarking suggests that List::Compare, at least in Regular mode, is considerably faster than Set::Scalar for those comparison methods which List::Compare makes available.

  Set::Bag enables one to deal more flexibly with the situation in which one has more than one instance of an element in a list.

- Set::Array – Arrays as objects with lots of handy methods (including set comparisons) and support for method chaining. (<http://search.cpan.org/dist/Set–Array/>)

  Set::Array, by Daniel Berger, now maintained by Ron Savage, "aims to provide built-in methods for operations that people are always asking how to do,and which already exist in languages like Ruby."

Among the many methods in this module are some for intersection, union, etc. To install Set::Array, you must first install the Want module, also available on CPAN.

ADDITIONAL CONTRIBUTORS

- Syohei YOSHIDA

  Pull request accepted May 22 2015.

- Paulo Custodio

  Pull request accepted June 07 2015, correcting errors in `_subset_subengine()`.

BUGS

There are no bug reports outstanding on List::Compare as of the most recent CPAN upload date of this distribution.

SUPPORT

Please report any bugs by mail to `bug-List-Compare@rt.cpan.org` or through the web interface at <http://rt.cpan.org>.

AUTHOR

James E. Keenan (jkeenan@cpan.org). When sending correspondence, please include 'List::Compare' or 'List−Compare' in your subject line.

Creation date: May 20, 2002. Last modification date: June 07 2015.

Development repository: <https://github.com/jkeenan/list−compare>

COPYRIGHT

DISCLAIMER OF WARRANTY