NAME

List::SomeUtils − Provide the stuff missing in List::Util

VERSION

version 0.58

SYNOPSIS

```
# import specific functions
use List::SomeUtils qw( any uniq );

if ( any {/foo/} uniq @has_duplicates ) {

    # do stuff
}

# import everything
use List::SomeUtils ':all';
```

DESCRIPTION

List::SomeUtils provides some trivial but commonly needed functionality on lists which is not going to go into List::Util.

All of the below functions are implementable in only a couple of lines of Perl code. Using the functions from this module however should give slightly better performance as everything is implemented in C. The pure-Perl implementation of these functions only serves as a fallback in case the C portions of this module couldn't be compiled on this machine.

WHY DOES THIS MODULE EXIST?

You might wonder why this module exists when we already have List::MoreUtils. In fact, this module is (nearly) the same code as is found in LMU with no significant changes. However, the LMU distribution depends on several modules for configuration (to run the Makefile.PL) that some folks in the Perl community don't think are appropriate for a module high upstream in the CPAN river.

I (Dave Rolsky) don't have a strong opinion on this, but I *do* like the functions provided by LMU, and I'm tired of getting patches and PRs to remove LMU from my code.

This distribution exists to let me use the functionality I like without having to get into tiring arguments about issues I don't really care about.

EXPORTS

Default behavior

Nothing by default. To import all of this module's symbols use the `:all` tag. Otherwise functions can be imported by name as usual:

```
use List::SomeUtils ':all';

use List::SomeUtils qw{ any firstidx };
```

Because historical changes to the API might make upgrading List::SomeUtils difficult for some projects, the legacy API is available via special import tags.

FUNCTIONS

Junctions

Treatment of an empty list

There are two schools of thought for how to evaluate a junction on an empty list:

•   Reduction to an identity (boolean)

•   Result is undefined (three-valued)

In the first case, the result of the junction applied to the empty list is determined by a mathematical reduction to an identity depending on whether the underlying comparison is ''or'' or ''and''. Conceptually:

```
                            "any are true"      "all are true"
                            --------------      --------------
        2 elements:         A || B || 0         A && B && 1
        1 element:          A || 0              A && 1
        0 elements:         0                   1
```

In the second case, three-value logic is desired, in which a junction applied to an empty list returns undef rather than true or false

Junctions with a _u suffix implement three-valued logic. Those without are boolean.

all BLOCK LIST

all_u BLOCK LIST

Returns a true value if all items in LIST meet the criterion given through BLOCK. Sets $_ for each item in LIST in turn:

```
  print "All values are non-negative"
    if all { $_ >= 0 } ($x, $y, $z);
```

For an empty LIST, all returns true (i.e. no values failed the condition) and all_u returns undef.

Thus, all_u(@list) is equivalent to @list ? all(@list) : undef.

Note: because Perl treats undef as false, you must check the return value of all_u with defined or you will get the opposite result of what you expect.

any BLOCK LIST

any_u BLOCK LIST

Returns a true value if any item in LIST meets the criterion given through BLOCK. Sets $_ for each item in LIST in turn:

```
  print "At least one non-negative value"
    if any { $_ >= 0 } ($x, $y, $z);
```

For an empty LIST, any returns false and any_u returns undef.

Thus, any_u(@list) is equivalent to @list ? any(@list) : undef.

none BLOCK LIST

none_u BLOCK LIST

Logically the negation of any. Returns a true value if no item in LIST meets the criterion given through BLOCK. Sets $_ for each item in LIST in turn:

```
  print "No non-negative values"
    if none { $_ >= 0 } ($x, $y, $z);
```

For an empty LIST, none returns true (i.e. no values failed the condition) and none_u returns undef.

Thus, none_u(@list) is equivalent to @list ? none(@list) : undef.

Note: because Perl treats undef as false, you must check the return value of none_u with defined or you will get the opposite result of what you expect.

notall BLOCK LIST

notall_u BLOCK LIST

Logically the negation of all. Returns a true value if not all items in LIST meet the criterion given through BLOCK. Sets $_ for each item in LIST in turn:

```
  print "Not all values are non-negative"
    if notall { $_ >= 0 } ($x, $y, $z);
```

For an empty LIST, notall returns false and notall_u returns undef.

Thus, `notall_u(@list)` is equivalent to `@list ? notall(@list) : undef`.

one BLOCK LIST

one_u BLOCK LIST

Returns a true value if precisely one item in LIST meets the criterion given through BLOCK. Sets `$_` for each item in LIST in turn:

```
print "Precisely one value defined"
    if one { defined($_) } @list;
```

Returns false otherwise.

For an empty LIST, `one` returns false and `one_u` returns `undef`.

The expression `one BLOCK LIST` is almost equivalent to `1 == true BLOCK LIST`, except for short-cutting. Evaluation of BLOCK will immediately stop at the second true value.

Transformation

apply BLOCK LIST

Makes a copy of the list and then passes each element *from the copy* to the BLOCK. Any changes or assignments to `$_` in the BLOCK will only affect the elements of the new list. However, if `$_` is a reference then changes to the referenced value will be seen in both the original and new list.

This function is similar to `map` but will not modify the elements of the input list:

```
my @list = (1 .. 4);
my @mult = apply { $_ *= 2 } @list;
print "\@list = @list\n";
print "\@mult = @mult\n";
__END__
@list = 1 2 3 4
@mult = 2 4 6 8
```

Think of it as syntactic sugar for

```
for (my @mult = @list) { $_ *= 2 }
```

Note that you must alter `$_` directly inside BLOCK in order for changes to make effect. New value returned from the BLOCK are ignored:

```
# @new is identical to @list.
my @new = apply { $_ * 2 } @list;

# @new is different from @list
my @new = apply { $_ =* 2 } @list;
```

insert_after BLOCK VALUE LIST

Inserts VALUE after the first item in LIST for which the criterion in BLOCK is true. Sets `$_` for each item in LIST in turn.

```
my @list = qw/This is a list/;
insert_after { $_ eq "a" } "longer" => @list;
print "@list";
__END__
This is a longer list
```

insert_after_string STRING VALUE LIST

Inserts VALUE after the first item in LIST which is equal to STRING.

```
  my @list = qw/This is a list/;
  insert_after_string "a", "longer" => @list;
  print "@list";
  __END__
  This is a longer list
```

pairwise BLOCK ARRAY1 ARRAY2

Evaluates BLOCK for each pair of elements in ARRAY1 and ARRAY2 and returns a new list consisting of
BLOCK's return values. The two elements are set to `$a` and `$b`. Note that those two are aliases to the
original value so changing them will modify the input arrays.

```
  @a = (1 .. 5);
  @b = (11 .. 15);
  @x = pairwise { $a + $b } @a, @b;      # returns 12, 14, 16, 18, 20

  # mesh with pairwise
  @a = qw/a b c/;
  @b = qw/1 2 3/;
  @x = pairwise { ($a, $b) } @a, @b;     # returns a, 1, b, 2, c, 3
```

mesh ARRAY1 ARRAY2 [ARRAY3 ...]

zip ARRAY1 ARRAY2 [ARRAY3 ...]

Returns a list consisting of the first elements of each array, then the second, then the third, etc, until all
arrays are exhausted.

Examples:

```
  @x = qw/a b c d/;
  @y = qw/1 2 3 4/;
  @z = mesh @x, @y;          # returns a, 1, b, 2, c, 3, d, 4

  @a = ('x');
  @b = ('1', '2');
  @c = qw/zip zap zot/;
  @d = mesh @a, @b, @c;    # x, 1, zip, undef, 2, zap, undef, undef, zot
```

`zip` is an alias for `mesh`.

uniq LIST

distinct LIST

Returns a new list by stripping duplicate values in LIST by comparing the values as hash keys, except that
undef is considered separate from ''. The order of elements in the returned list is the same as in LIST. In
scalar context, returns the number of unique elements in LIST.

```
  my @x = uniq 1, 1, 2, 2, 3, 5, 3, 4; # returns 1 2 3 5 4
  my $x = uniq 1, 1, 2, 2, 3, 5, 3, 4; # returns 5
  # returns "Mike", "Michael", "Richard", "Rick"
  my @n = distinct "Mike", "Michael", "Richard", "Rick", "Michael", "Rick"
  # returns '', undef, 'S1', A5'
  my @s = distinct '', undef, 'S1', 'A5'
  # returns '', undef, 'S1', A5'
  my @w = uniq undef, '', 'S1', 'A5'
```

`distinct` is an alias for `uniq`.

RT#49800 can be used to give feedback about this behavior.

singleton

Returns a new list by stripping values in LIST occurring more than once by comparing the values as hash

keys, except that undef is considered separate from ''. The order of elements in the returned list is the same as in LIST. In scalar context, returns the number of elements occurring only once in LIST.

```
my @x = singleton 1,1,2,2,3,4,5 # returns 3 4 5
```

Partitioning

after BLOCK LIST

Returns a list of the values of LIST after (and not including) the point where BLOCK returns a true value. Sets $_ for each element in LIST in turn.

```
@x = after { $_ % 5 == 0 } (1..9);     # returns 6, 7, 8, 9
```

after_incl BLOCK LIST

Same as `after` but also includes the element for which BLOCK is true.

before BLOCK LIST

Returns a list of values of LIST up to (and not including) the point where BLOCK returns a true value. Sets $_ for each element in LIST in turn.

before_incl BLOCK LIST

Same as `before` but also includes the element for which BLOCK is true.

part BLOCK LIST

Partitions LIST based on the return value of BLOCK which denotes into which partition the current value is put.

Returns a list of the partitions thusly created. Each partition created is a reference to an array.

```
my $i = 0;
my @part = part { $i++ % 2 } 1 .. 8;    # returns [1, 3, 5, 7], [2, 4, 6, 8]
```

You can have a sparse list of partitions as well where non-set partitions will be undef:

```
my @part = part { 2 } 1 .. 10;                # returns undef, undef, [ 1 .. 10 ]
```

Be careful with negative values, though:

```
my @part = part { -1 } 1 .. 10;
__END__
Modification of non-creatable array value attempted, subscript -1 ...
```

Negative values are only ok when they refer to a partition previously created:

```
my @idx  = ( 0, 1, -1 );
my $i    = 0;
my @part = part { $idx[$i++ % 3] } 1 .. 8; # [1, 4, 7], [2, 3, 5, 6, 8]
```

Iteration

each_array ARRAY1 ARRAY2 ...

Creates an array iterator to return the elements of the list of arrays ARRAY1, ARRAY2 throughout ARRAYn in turn. That is, the first time it is called, it returns the first element of each array. The next time, it returns the second elements. And so on, until all elements are exhausted.

This is useful for looping over more than one array at once:

```
my $ea = each_array(@a, @b, @c);
while ( my ($a, $b, $c) = $ea->() )   { .... }
```

The iterator returns the empty list when it reached the end of all arrays.

If the iterator is passed an argument of 'index', then it returns the index of the last fetched set of values, as a scalar.

each_arrayref LIST

Like each_array, but the arguments are references to arrays, not the plain arrays.

natatime EXPR, LIST

Creates an array iterator, for looping over an array in chunks of $n items at a time. (n at a time, get it?). An example is probably a better explanation than I could give in words.

Example:

```
my @x = ('a' .. 'g');
my $it = natatime 3, @x;
while (my @vals = $it->())
{
  print "@vals\n";
}
```

This prints

```
a b c
d e f
g
```

Searching

bsearch BLOCK LIST

Performs a binary search on LIST which must be a sorted list of values. BLOCK must return a negative value if the current element (stored in $_) is smaller, a positive value if it is bigger and zero if it matches.

Returns a boolean value in scalar context. In list context, it returns the element if it was found, otherwise the empty list.

bsearchidx BLOCK LIST

bsearch_index BLOCK LIST

Performs a binary search on LIST which must be a sorted list of values. BLOCK must return a negative value if the current element (stored in $_) is smaller, a positive value if it is bigger and zero if it matches.

Returns the index of found element, otherwise −1.

bsearch_index is an alias for bsearchidx.

firstval BLOCK LIST

first_value BLOCK LIST

Returns the first element in LIST for which BLOCK evaluates to true. Each element of LIST is set to $_ in turn. Returns undef if no such element has been found.

first_value is an alias for firstval.

onlyval BLOCK LIST

only_value BLOCK LIST

Returns the only element in LIST for which BLOCK evaluates to true. Sets $_ for each item in LIST in turn. Returns undef if no such element has been found.

only_value is an alias for onlyval.

lastval BLOCK LIST

last_value BLOCK LIST

Returns the last value in LIST for which BLOCK evaluates to true. Each element of LIST is set to $_ in turn. Returns undef if no such element has been found.

last_value is an alias for lastval.

firstres BLOCK LIST

first_result BLOCK LIST

Returns the result of BLOCK for the first element in LIST for which BLOCK evaluates to true. Each element of LIST is set to `$_` in turn. Returns `undef` if no such element has been found.

`first_result` is an alias for `firstres`.

onlyres BLOCK LIST

only_result BLOCK LIST

Returns the result of BLOCK for the first element in LIST for which BLOCK evaluates to true. Sets `$_` for each item in LIST in turn. Returns `undef` if no such element has been found.

`only_result` is an alias for `onlyres`.

lastres BLOCK LIST

last_result BLOCK LIST

Returns the result of BLOCK for the last element in LIST for which BLOCK evaluates to true. Each element of LIST is set to `$_` in turn. Returns `undef` if no such element has been found.

`last_result` is an alias for `lastres`.

indexes BLOCK LIST

Evaluates BLOCK for each element in LIST (assigned to `$_`) and returns a list of the indices of those elements for which BLOCK returned a true value. This is just like `grep` only that it returns indices instead of values:

```
@x = indexes { $_ % 2 == 0 } (1..10);    # returns 1, 3, 5, 7, 9
```

firstidx BLOCK LIST

first_index BLOCK LIST

Returns the index of the first element in LIST for which the criterion in BLOCK is true. Sets `$_` for each item in LIST in turn:

```
my @list = (1, 4, 3, 2, 4, 6);
printf "item with index %i in list is 4", firstidx { $_ == 4 } @list;
__END__
item with index 1 in list is 4
```

Returns −1 if no such item could be found.

`first_index` is an alias for `firstidx`.

onlyidx BLOCK LIST

only_index BLOCK LIST

Returns the index of the only element in LIST for which the criterion in BLOCK is true. Sets `$_` for each item in LIST in turn:

```
my @list = (1, 3, 4, 3, 2, 4);
printf "uniqe index of item 2 in list is %i", onlyidx { $_ == 2 } @list;
__END__
unique index of item 2 in list is 4
```

Returns −1 if either no such item or more than one of these has been found.

`only_index` is an alias for `onlyidx`.

lastidx BLOCK LIST

last_index BLOCK LIST

Returns the index of the last element in LIST for which the criterion in BLOCK is true. Sets `$_` for each item in LIST in turn:

```
    my @list = (1, 4, 3, 2, 4, 6);
    printf "item with index %i in list is 4", lastidx { $_ == 4 } @list;
    __END__
    item with index 4 in list is 4
```

Returns −1 if no such item could be found.

`last_index` is an alias for `lastidx`.

Sorting

sort_by BLOCK LIST

Returns the list of values sorted according to the string values returned by the KEYFUNC block or function. A typical use of this may be to sort objects according to the string value of some accessor, such as

```
    sort_by { $_->name } @people
```

The key function is called in scalar context, being passed each value in turn as both `$_` and the only argument in the parameters, `@_`. The values are then sorted according to string comparisons on the values returned. This is equivalent to

```
    sort { $a->name cmp $b->name } @people
```

except that it guarantees the name accessor will be executed only once per value. One interesting use-case is to sort strings which may have numbers embedded in them ''naturally'', rather than lexically.

```
    sort_by { s/(\d+)/sprintf "%09d", $1/eg; $_ } @strings
```

This sorts strings by generating sort keys which zero-pad the embedded numbers to some level (9 digits in this case), helping to ensure the lexical sort puts them in the correct order.

nsort_by BLOCK LIST

Similar to sort_by but compares its key values numerically.

Counting and calculation

true BLOCK LIST

Counts the number of elements in LIST for which the criterion in BLOCK is true. Sets `$_` for each item in LIST in turn:

```
    printf "%i item(s) are defined", true { defined($_) } @list;
```

false BLOCK LIST

Counts the number of elements in LIST for which the criterion in BLOCK is false. Sets `$_` for each item in LIST in turn:

```
    printf "%i item(s) are not defined", false { defined($_) } @list;
```

minmax LIST

Calculates the minimum and maximum of LIST and returns a two element list with the first element being the minimum and the second the maximum. Returns the empty list if LIST was empty.

The `minmax` algorithm differs from a naive iteration over the list where each element is compared to two values being the so far calculated min and max value in that it only requires $3n/2 - 2$ comparisons. Thus it is the most efficient possible algorithm.

However, the Perl implementation of it has some overhead simply due to the fact that there are more lines of Perl code involved. Therefore, LIST needs to be fairly big in order for `minmax` to win over a naive implementation. This limitation does not apply to the XS version.

mode LIST

Calculates the most common items in the list and returns them as a list. This is effectively done by string comparisons, so references will be stringified. If they implement string overloading, this will be used.

If more than one item appears the same number of times in the list, all such items will be returned. For

example, the mode of a unique list is the list itself.

This function returns a list in list context. In scalar context it returns a count indicating the number of modes in the list.

MAINTENANCE

The maintenance goal is to preserve the documented semantics of the API; bug fixes that bring actual behavior in line with semantics are allowed. New API functions may be added over time. If a backwards incompatible change is unavoidable, we will attempt to provide support for the legacy API using the same export tag mechanism currently in place.

This module attempts to use few non-core dependencies. Non-core configuration and testing modules will be bundled when reasonable; run-time dependencies will be added only if they deliver substantial benefit.

KNOWN ISSUES

There is a problem with a bug in 5.6.x perls. It is a syntax error to write things like:

```
my @x = apply { s/foo/bar/ } qw{ foo bar baz };
```

It has to be written as either

```
my @x = apply { s/foo/bar/ } 'foo', 'bar', 'baz';
```

or

```
my @x = apply { s/foo/bar/ } my @dummy = qw/foo bar baz/;
```

Perl 5.5.x and Perl 5.8.x don't suffer from this limitation.

If you have a functionality that you could imagine being in this module, please drop me a line. This module's policy will be less strict than List::Util's when it comes to additions as it isn't a core module.

When you report bugs, it would be nice if you could additionally give me the output of your program with the environment variable LIST_MOREUTILS_PP set to a true value. That way I know where to look for the problem (in XS, pure-Perl or possibly both).

THANKS

Tassilo von Parseval

Credits go to a number of people: Steve Purkis for giving me namespace advice and James Keenan and Terrence Branno for their effort of keeping the CPAN tidier by making List::Util obsolete.

Brian McCauley suggested the inclusion of **apply()** and provided the pure-Perl implementation for it.

Eric J. Roode asked me to add all functions from his module List::SomeUtil into this one. With minor modifications, the pure-Perl implementations of those are by him.

The bunch of people who almost immediately pointed out the many problems with the glitchy 0.07 release (Slaven Rezic, Ron Savage, CPAN testers).

A particularly nasty memory leak was spotted by Thomas A. Lowery.

Lars Thegler made me aware of problems with older Perl versions.

Anno Siegel de-orphaned **each_arrayref()**.

David Filmer made me aware of a problem in each_arrayref that could ultimately lead to a segfault.

Ricardo Signes suggested the inclusion of **part()** and provided the Perl-implementation.

Robin Huston kindly fixed a bug in perl's MULTICALL API to make the XS-implementation of **part()** work.

Jens Rehsack

Credits goes to all people contributing feedback during the v0.400 development releases.

Special thanks goes to David Golden who spent a lot of effort to develop a design to support current state of CPAN as well as ancient software somewhere in the dark. He also contributed a lot of patches to refactor the API frontend to welcome any user of List::SomeUtils – from ancient past to recently last used.

Toby Inkster provided a lot of useful feedback for sane importer code and was a nice sounding board for

API discussions.

Peter Rabbitson provided a sane git repository setup containing entire package history.

TODO

A pile of requests from other people is still pending further processing in my mailbox. This includes:

- List::Util export pass-through

  Allow **List::SomeUtils** to pass-through the regular List::Util functions to end users only need to `use` the one module.

- uniq_by(&@)

  Use code-reference to extract a key based on which the uniqueness is determined. Suggested by Aaron Crane.

- delete_index

- random_item

- random_item_delete_index

- list_diff_hash

- list_diff_inboth

- list_diff_infirst

- list_diff_insecond

  These were all suggested by Dan Muey.

- listify

  Always return a flat list when either a simple scalar value was passed or an array-reference. Suggested by Mark Summersault.

SEE ALSO

List::Util, List::AllUtils, List::UtilsBy

HISTORICAL COPYRIGHT

Some parts copyright 2011 Aaron Crane.

Copyright 2004 − 2010 by Tassilo von Parseval

Copyright 2013 − 2015 by Jens Rehsack

SUPPORT

Bugs may be submitted at <https://github.com/houseabsolute/List−SomeUtils/issues>.

I am also usually active on IRC as 'autarch' on `irc://irc.perl.org`.

SOURCE

The source code repository for List-SomeUtils can be found at <https://github.com/houseabsolute/List−SomeUtils>.

DONATIONS

If you'd like to thank me for the work I've done on this module, please consider making a ''donation'' to me via PayPal. I spend a lot of free time creating free software, and would appreciate any support you'd care to offer.

Please note that **I am not suggesting that you must do this** in order for me to continue working on this particular software. I will continue to do so, inasmuch as I have in the past, for as long as it interests me.

Similarly, a donation made in this way will probably not make me work on this software much more, unless I get so many donations that I can consider working on free software full time (let's all have a chuckle at that together).

To donate, log into PayPal and send money to autarch@urth.org, or use the button at

<http://www.urth.org/˜autarch/fs−donation.html>.

AUTHORS

- Tassilo von Parseval <tassilo.von.parseval@rwth−aachen.de>

- Adam Kennedy <adamk@cpan.org>

- Jens Rehsack <rehsack@cpan.org>

- Dave Rolsky <autarch@urth.org>

CONTRIBUTORS

- Aaron Crane <arc@cpan.org>

- BackPan <BackPan>

- bay−max1 <34803732+bay−max1@users.noreply.github.com>

- Brad Forschinger <bnjf@bnjf.id.au>

- David Golden <dagolden@cpan.org>

- jddurand <jeandamiendurand@free.fr>

- Jens Rehsack <sno@netbsd.org>

- J.R. Mash <jrmash@cpan.org>

- Karen Etheridge <ether@cpan.org>

- Ricardo Signes <rjbs@cpan.org>

- Toby Inkster <mail@tobyinkster.co.uk>

- Tokuhiro Matsuno <tokuhirom@cpan.org>

- Tom Wyant <wyant@cpan.org>

COPYRIGHT AND LICENSE

This software is copyright (c) 2019 by Dave Rolsky <autarch@urth.org>.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

The full text of the license can be found in the *LICENSE* file included with this distribution.