Nix (builtins) & Nixpkgs (lib) Functions

Contents

Nix Builtin Functions

builtins.zipAttrsWith

f, list

Transpose a list of attribute sets into an attribute set of lists, then apply `mapAttrs`.

`f` receives two arguments: the attribute name and a non-empty list of all values encountered for that attribute name.

The result is an attribute set where the attribute names are the union of the attribute names in each element of `list`. The attribute values are the return values of `f`.

```
builtins.zipAttrsWith
  (name: values: { inherit name values; })
  [ { a = "x"; } { a = "y"; b = "z"; } ]
```

evaluates to

```
{
  a = { name = "a"; values = [ "x" "y" ]; };
  b = { name = "b"; values = [ "z" ]; };
}
```

builtins.typeOf

e

Return a string representing the type of the value *e*, namely `"int"`, `"bool"`, `"string"`, `"path"`, `"null"`, `"set"`, `"list"`, `"lambda"` or `"float"`.

builtins.tryEval

e

Try to shallowly evaluate *e*. Return a set containing the attributes `success` (`true` if *e* evaluated successfully, `false` if an error was thrown) and `value`, equalling *e* if successful and `false` otherwise. `tryEval` will only prevent errors created by `throw` or `assert` from being thrown. Errors `tryEval` will not catch are for example those created by `abort` and type errors generated by builtins. Also note that this doesn't evaluate *e* deeply, so `let e = { x = throw ""; }; in (builtins.tryEval e).success` will be `true`. Using `builtins.deepSeq` one can get the expected result: `let e = { x = throw ""; }; in (builtins.tryEval (builtins.deepSeq e e)).success` will be `false`.

builtins.trace

e1, *e2*

Evaluate *e1* and print its abstract syntax representation on standard error. Then return *e2*. This function is useful for debugging.

builtins.toXML

e

Return a string containing an XML representation of *e*. The main application for `toXML` is to communicate information with the builder in a more structured format than plain environment variables.

Here is an example where this is the case:

```
{ stdenv, fetchurl, libxslt, jira, uberwiki }:

stdenv.mkDerivation (rec {
  name = "web-server";

  buildInputs = [ libxslt ];

  builder = builtins.toFile "builder.sh" "
    source $stdenv/setup
    mkdir $out
    echo "$servlets" | xsltproc ${stylesheet} - >
$out/server-conf.xml ①
  ";

  stylesheet = builtins.toFile "stylesheet.xsl" ②
   "<?xml version='1.0' encoding='UTF-8'?>
    <xsl:stylesheet xmlns:xsl='http://www.w3.org/1999/XSL
/Transform' version='1.0'>
      <xsl:template match='/'>
        <Configure>
          <xsl:for-each select='/expr/list/attrs'>
            <Call name='addWebApplication'>
              <Arg><xsl:value-of select=\"attr[@name =
'path']/string/@value\" /></Arg>
              <Arg><xsl:value-of select=\"attr[@name =
'war']/path/@value\" /></Arg>
            </Call>
          </xsl:for-each>
        </Configure>
      </xsl:template>
    </xsl:stylesheet>
  ";

  servlets = builtins.toXML [ ③
    { path = "/bugtracker"; war = jira + "/lib/atlassian-
jira.war"; }
    { path = "/wiki"; war = uberwiki + "/uberwiki.war"; }
```

```
    ];
  })
```

The builder is supposed to generate the configuration file for a Jetty servlet container. A servlet container contains a number of servlets (`*.war` files) each exported under a specific URI prefix. So the servlet configuration is a list of sets containing the `path` and `war` of the servlet (①). This kind of information is difficult to communicate with the normal method of passing information through an environment variable, which just concatenates everything together into a string (which might just work in this case, but wouldn't work if fields are optional or contain lists themselves). Instead the Nix expression is converted to an XML representation with `toXML`, which is unambiguous and can easily be processed with the appropriate tools. For instance, in the example an XSLT stylesheet (at point ②) is applied to it (at point ①) to generate the XML configuration file for the Jetty server. The XML representation produced at point ③ by `toXML` is as follows:

```
<?xml version='1.0' encoding='utf-8'?>
<expr>
  <list>
    <attrs>
      <attr name="path">
        <string value="/bugtracker" />
      </attr>
      <attr name="war">
        <path value="/nix/store/d1jh9pasa7k2...-jira/lib
/atlassian-jira.war" />
      </attr>
    </attrs>
    <attrs>
      <attr name="path">
        <string value="/wiki" />
      </attr>
      <attr name="war">
        <path value="/nix/store/y6423b1yi4sx...-uberwiki
/uberwiki.war" />
      </attr>
    </attrs>
  </list>
</expr>
```

Note that we used the `toFile` built-in to write the builder and the stylesheet "inline" in the Nix expression. The path of the stylesheet is spliced into the builder using the syntax `xsltproc ${stylesheet}`.

builtins.toString

e

Convert the expression *e* to a string. *e* can be:

- A string (in which case the string is returned unmodified).

- A path (e.g., `toString /foo/bar` yields `"/foo/bar"`.
- A set containing `{ __toString = self: ...; }` or `{ outPath = ...; }`.
- An integer.
- A list, in which case the string representations of its elements are joined with spaces.
- A Boolean (`false` yields `""`, `true` yields `"1"`).
- `null`, which yields the empty string.

builtins.toPath

s

DEPRECATED. Use `/. + "/path"` to convert a string into an absolute path. For relative paths, use `./. + "/path"`.

builtins.toJSON

e

Return a string containing a JSON representation of *e*. Strings, integers, floats, booleans, nulls and lists are mapped to their JSON equivalents. Sets (except derivations) are represented as objects. Derivations are translated to a JSON string containing the derivation's output path. Paths are copied to the store and represented as a JSON string of the resulting store path.

builtins.toFile

name, s

Store the string *s* in a file in the Nix store and return its path. The file has suffix *name*. This file can be used as an input to derivations.

One application is to write builders “inline”. For instance, the following Nix expression combines the Nix expression for GNU Hello and its build script into one file:

```
{ stdenv, fetchurl, perl }:

stdenv.mkDerivation {
  name = "hello-2.1.1";

  builder = builtins.toFile "builder.sh" "
    source $stdenv/setup

    PATH=$perl/bin:$PATH

    tar xvfz $src
    cd hello-*
    ./configure --prefix=$out
    make
    make install
  ";

  src = fetchurl {
    url = "http://ftp.nluug.nl/pub/gnu/hello/hello-
2.1.1.tar.gz";
    sha256 =
"1md7jsfd8pa45z73bz1kszpp01yw6x5ljkjk2hx7wl800any6465";
  };
  inherit perl;
}
```

It is even possible for one file to refer to another, e.g.,

```
builder = let
  configFile = builtins.toFile "foo.conf" "
    # This is some dummy configuration file.
    ...
  ";
in builtins.toFile "builder.sh" "
  source $stdenv/setup
  ...
  cp ${configFile} $out/etc/foo.conf
";
```

Note that `${configFile}` is an antiquotation, so the result of the expression `configFile` (i.e., a path like `/nix/store/m7p7jfny445k...-foo.conf`) will be spliced into the resulting string.

It is however *not* allowed to have files mutually referring to each other, like so:

```
let
  foo = builtins.toFile "foo" "...${bar}...";
  bar = builtins.toFile "bar" "...${foo}...";
in foo
```

This is not allowed because it would cause a cyclic dependency in the computation of the cryptographic hashes for `foo` and `bar`.

It is also not possible to reference the result of a derivation. If you are using Nixpkgs, the `writeTextFile` function is able to do that.

builtins.throw

s

Throw an error message *s*. This usually aborts Nix expression evaluation, but in `nix-env -qa` and other commands that try to evaluate a set of derivations to get information about those

derivations, a derivation that throws an error is silently skipped (which is not the case for `abort`).

builtins.tail

list

Return the second to last elements of a list; abort evaluation if the argument isn't a list or is an empty list.

> ***Warning***
>
> *This function should generally be avoided since it's inefficient: unlike Haskell's `tail`, it takes O(n) time, so recursing over a list by repeatedly calling `tail` takes O(n^2) time.*

builtins.substring

start, len, s

Return the substring of *s* from character position *start* (zero-based) up to but not including *start + len*. If *start* is greater than the length of the string, an empty string is returned, and if *start + len* lies beyond the end of the string, only the substring up to the end of the string is returned. *start* must be non-negative. For example,

```
builtins.substring 0 3 "nixos"
```

evaluates to `"nix"`.

builtins.sub

e1, e2

Return the difference between the numbers *e1* and *e2*.

builtins.stringLength

e

Return the length of the string *e*. If *e* is not a string, evaluation is aborted.

builtins.storePath

path

This function allows you to define a dependency on an already existing store path. For example, the derivation attribute `src = builtins.storePath /nix/store/f1d18v1y…-source` causes the derivation to depend on the specified path, which must exist or be substitutable. Note that this differs from a plain path (e.g. `src = /nix/store/f1d18v1y…-source`) in that the latter causes the path to be *copied* again to the Nix store, resulting in a new path (e.g. `/nix/store/ld01dnzc…-source-source`).

This function is not available in pure evaluation mode.

builtins.splitVersion

s

Split a string representing a version into its components, by the same version splitting logic underlying the version comparison in `nix-env -u`.

builtins.split

regex, str

Returns a list composed of non matched strings interleaved with the lists of the extended POSIX regular expression *regex* matches of *str*. Each item in the lists of matched sequences is a regex group.

```
builtins.split "(a)b" "abc"
```

Evaluates to `[ "" [ "a" ] "c" ]`.

```
builtins.split "([ac])" "abc"
```

Evaluates to `[ "" [ "a" ] "b" [ "c" ] "" ]`.

```
builtins.split "(a)|(c)" "abc"
```

Evaluates to `[ "" [ "a" null ] "b" [ null "c" ] "" ]`.

```
builtins.split "([[:upper:]]+)" "  FOO   "
```

Evaluates to `[ " " [ "FOO" ] " " ]`.

builtins.sort

comparator, *list*

Return *list* in sorted order. It repeatedly calls the function *comparator* with two elements. The comparator should return `true` if the first element is less than the second, and `false` otherwise. For example,

```
builtins.sort builtins.lessThan [ 483 249 526 147 42 77 ]
```

produces the list `[ 42 77 147 249 483 526 ]`.

This is a stable sort: it preserves the relative order of elements deemed equal by the comparator.

builtins.seq

e1, *e2*

Evaluate *e1*, then evaluate and return *e2*. This ensures that a computation is strict in the value of *e1*.

builtins.replaceStrings

from, *to*, *s*

Given string *s*, replace every occurrence of the strings in *from* with the corresponding string in *to*. For example,

```
builtins.replaceStrings ["oo" "a"] ["a" "i"] "foobar"
```

evaluates to `"fabir"`.

builtins.removeAttrs

set, list

Remove the attributes listed in *list* from *set*. The attributes don't have to exist in *set*. For instance,

```
removeAttrs { x = 1; y = 2; z = 3; } [ "a" "x" "z" ]
```

evaluates to `{ y = 2; }`.

builtins.readFile

path

Return the contents of the file *path* as a string.

builtins.readDir

path

Return the contents of the directory *path* as a set mapping directory entries to the corresponding file type. For instance, if directory `A` contains a regular file `B` and another directory `C`, then `builtins.readDir ./A` will return the set

```
{ B = "regular"; C = "directory"; }
```

The possible values for the file type are `"regular"`, `"directory"`, `"symlink"` and `"unknown"`.

builtins.placeholder

output

Return a placeholder string for the specified *output* that will be substituted by the corresponding output path at build time. Typical outputs would be `"out"`, `"bin"` or `"dev"`.

builtins.pathExists

path

Return `true` if the path *path* exists at evaluation time, and `false` otherwise.

builtins.path

args

An enrichment of the built-in path type, based on the attributes present in *args*. All are optional except `path`:

- path
  The underlying path.

- name
  The name of the path when added to the store. This can used to reference paths that have nix-illegal characters in their names, like `@`.

- filter
  A function of the type expected by `builtins.filterSource`, with the same semantics.

- recursive
  When `false`, when `path` is added to the store it is with a flat hash, rather than a hash of the NAR serialization of the file. Thus, `path` must refer to a regular file, not a directory. This

allows similar behavior to `fetchurl`. Defaults to `true`.

- sha256
  When provided, this is the expected hash of the file at the path. Evaluation will fail if the hash is incorrect, and providing a hash allows `builtins.path` to be used even when the `pure-eval` nix config option is on.

builtins.partition

pred, list

Given a predicate function *pred*, this function returns an attrset containing a list named `right`, containing the elements in *list* for which *pred* returned `true`, and a list named `wrong`, containing the elements for which it returned `false`. For example,

```
builtins.partition (x: x > 10) [1 23 9 3 42]
```

evaluates to

```
{ right = [ 23 42 ]; wrong = [ 1 9 3 ]; }
```

builtins.parseDrvName

s

Split the string *s* into a package name and version. The package name is everything up to but not including the first dash followed by a digit, and the version is everything following that dash. The result is returned in a set `{ name, version }`. Thus, `builtins.parseDrvName "nix-0.12pre12876"` returns `{ name = "nix"; version = "0.12pre12876"; }`.

builtins.mul

e1, e2

Return the product of the numbers *e1* and *e2*.

builtins.match

regex, *str*

Returns a list if the extended POSIX regular expression *regex* matches *str* precisely, otherwise returns `null`. Each item in the list is a regex group.

```
builtins.match "ab" "abc"
```

Evaluates to `null`.

```
builtins.match "abc" "abc"
```

Evaluates to `[ ]`.

```
builtins.match "a(b)(c)" "abc"
```

Evaluates to `[ "b" "c" ]`.

```
builtins.match "[[:space:]]+([[:upper:]]+)[[:space:]]+" "
FOO   "
```

Evaluates to `[ "foo" ]`.

builtins.mapAttrs

f, *attrset*

Apply function *f* to every element of *attrset*. For example,

```
builtins.mapAttrs (name: value: value * 10) { a = 1; b =
2; }
```

evaluates to `{ a = 10; b = 20; }`.

builtins.map

f, list

Apply the function *f* to each element in the list *list*. For example,

```
map (x: "foo" + x) [ "bar" "bla" "abc" ]
```

evaluates to `[ "foobar" "foobla" "fooabc" ]`.

builtins.listToAttrs

e

Construct a set from a list specifying the names and values of each attribute. Each element of the list should be a set consisting of a string-valued attribute `name` specifying the name of the attribute, and an attribute `value` specifying its value. Example:

```
builtins.listToAttrs
  [ { name = "foo"; value = 123; }
    { name = "bar"; value = 456; }
  ]
```

evaluates to

```
{ foo = 123; bar = 456; }
```

builtins.lessThan

e1, e2

Return `true` if the number *e1* is less than the number *e2*, and `false` otherwise. Evaluation aborts if either *e1* or *e2* does not evaluate to a number.

builtins.length

e

Return the length of the list *e*.

builtins.isString

e

Return `true` if *e* evaluates to a string, and `false` otherwise.

builtins.isPath

e

Return `true` if *e* evaluates to a path, and `false` otherwise.

builtins.isNull

e

Return `true` if *e* evaluates to `null`, and `false` otherwise.

> ***Warning***
>
> *This function is* deprecated*; just write* `e == null` *instead.*

builtins.isList

e

Return `true` if *e* evaluates to a list, and `false` otherwise.

builtins.isInt

e

Return `true` if *e* evaluates to an integer, and `false` otherwise.

builtins.isFunction

e

Return `true` if *e* evaluates to a function, and `false` otherwise.

builtins.isFloat

e

Return `true` if *e* evaluates to a float, and `false` otherwise.

builtins.isBool

e

Return `true` if *e* evaluates to a bool, and `false` otherwise.

builtins.isAttrs

e

Return `true` if *e* evaluates to a set, and `false` otherwise.

builtins.intersectAttrs

e1, e2

Return a set consisting of the attributes in the set *e2* that also exist in the set *e1*.

builtins.import

path

Load, parse and return the Nix expression in the file *path*. If *path* is a directory, the file `default.nix` in that directory is loaded. Evaluation aborts if the file doesn't exist or contains an incorrect Nix expression. `import` implements Nix's module system: you can put any Nix expression (such as a set or a function) in a separate file, and use it from Nix expressions in other files.

> ***Note***

> *Unlike some languages,* `import` *is a regular function in Nix. Paths using the angle bracket syntax (e.g.,* `import` <foo>*) are normal path values.*

A Nix expression loaded by `import` must not contain any *free variables* (identifiers that are not defined in the Nix expression itself and are not built-in). Therefore, it cannot refer to variables that are in scope at the call site. For instance, if you have a calling expression

```
rec {
  x = 123;
  y = import ./foo.nix;
}
```

then the following `foo.nix` will give an error:

```
x + 456
```

since `x` is not in scope in `foo.nix`. If you want `x` to be available in `foo.nix`, you should pass it as a function argument:

```
rec {
  x = 123;
  y = import ./foo.nix x;
}
```

and

```
x: x + 456
```

(The function argument doesn't have to be called `x` in `foo.nix`; any name would work.)

builtins.head

list

Return the first element of a list; abort evaluation if the argument isn't a list or is an empty list. You can test whether a list is empty by comparing it with `[]`.

builtins.hashString

type, *s*

Return a base-16 representation of the cryptographic hash of string *s*. The hash algorithm specified by *type* must be one of `"md5"`, `"sha1"`, `"sha256"` or `"sha512"`.

builtins.hashFile

type, *p*

Return a base-16 representation of the cryptographic hash of the file at path *p*. The hash algorithm specified by *type* must be one of `"md5"`, `"sha1"`, `"sha256"` or `"sha512"`.

builtins.hasAttr

s, *set*

`hasAttr` returns `true` if *set* has an attribute named *s*, and `false` otherwise. This is a dynamic version of the `?` operator, since *s* is an expression rather than an identifier.

builtins.groupBy

f, *list*

Groups elements of *list* together by the string returned from the function *f* called on each element. It returns an attribute set where each attribute value contains the elements of *list* that are mapped to the same corresponding attribute name returned by *f*.

For example,

```
builtins.groupBy (builtins.substring 0 1) ["foo" "bar"
"baz"]
```

evaluates to

```
{ b = [ "bar" "baz" ]; f = [ "foo" ]; }
```

builtins.getFlake

args

Fetch a flake from a flake reference, and return its output attributes and some metadata. For example:

```
(builtins.getFlake
"nix/55bc52401966fbffa525c574c14f67b00bc4fb3a").packages.x86_64-
linux.nix
```

Unless impure evaluation is allowed (`--impure`), the flake reference must be "locked", e.g. contain a Git revision or content hash. An example of an unlocked usage is:

```
(builtins.getFlake "github:edolstra/dwarffs").rev
```

This function is only available if you enable the experimental feature `flakes`.

builtins.getEnv

s

`getEnv` returns the value of the environment variable *s*, or an empty string if the variable doesn't exist. This function should be used with care, as it can introduce all sorts of nasty environment dependencies in your Nix expression.

`getEnv` is used in Nix Packages to locate the file `~/.nixpkgs`

`/config.nix`, which contains user-local settings for Nix Packages. (That is, it does a `getEnv "HOME"` to locate the user's home directory.)

builtins.getAttr

s, *set*

`getAttr` returns the attribute named *s* from *set*. Evaluation aborts if the attribute doesn't exist. This is a dynamic version of the `.` operator, since *s* is an expression rather than an identifier.

builtins.genericClosure

attrset

Take an *attrset* with values named `startSet` and `operator` in order to return a *list of attrsets* by starting with the `startSet`, recursively applying the `operator` function to each element. The *attrsets* in the `startSet` and produced by the `operator` must each contain value named `key` which are comparable to each other. The result is produced by repeatedly calling the operator for each element encountered with a unique key, terminating when no new elements are produced. For example,

```
builtins.genericClosure {
  startSet = [ {key = 5;} ];
  operator = item: [{
    key = if (item.key / 2 ) * 2 == item.key
         then item.key / 2
         else 3 * item.key + 1;
  }];
}
```

evaluates to

```
[ { key = 5; } { key = 16; } { key = 8; } { key = 4; } {
key = 2; } { key = 1; } ]
```

builtins.genList

generator, length

Generate list of size *length*, with each element *i* equal to the value returned by *generator* `i`. For example,

```
builtins.genList (x: x * x) 5
```

returns the list `[ 0 1 4 9 16 ]`.

builtins.functionArgs

f

Return a set containing the names of the formal arguments expected by the function *f*. The value of each attribute is a Boolean denoting whether the corresponding argument has a default value. For instance, `functionArgs ({ x, y ? 123}: ...) = { x = false; y = true; }`.

"Formal argument" here refers to the attributes pattern-matched by the function. Plain lambdas are not included, e.g. `functionArgs (x: ...) = { }`.

builtins.fromJSON

e

Convert a JSON string to a Nix value. For example,

```
builtins.fromJSON ''{"x": [1, 2, 3], "y": null}''
```

returns the value `{ x = [ 1 2 3 ]; y = null; }`.

builtins.foldl'

op, nul, list

Reduce a list by applying a binary operator, from left to right, e.g. `foldl' op nul [x0 x1 x2 ...] = op (op (op nul x0) x1) x2) ...`. The operator is applied strictly, i.e., its arguments are evaluated first. For example, `foldl' (x: y: x + y) 0 [1 2 3]` evaluates to 6.

builtins.floor

double

Converts an IEEE-754 double-precision floating-point number (*double*) to the next lower integer.

If the datatype is neither an integer nor a “float”, an evaluation error will be thrown.

builtins.filterSource

e1, e2

> ***Warning***
>
> *`filterSource` should not be used to filter store paths. Since `filterSource` uses the name of the input directory while naming the output directory, doing so will produce a directory name in the form of `<hash2>-<hash>-<name>`, where `<hash>-<name>` is the name of the input directory. Since `<hash>` depends on the unfiltered directory, the name of the output directory will indirectly depend on files that are filtered out by the function. This will trigger a rebuild even when a filtered out file is changed. Use `builtins.path` instead, which allows specifying the name of the output directory.*

This function allows you to copy sources into the Nix store while

filtering certain files. For instance, suppose that you want to use the directory `source-dir` as an input to a Nix expression, e.g.

```
stdenv.mkDerivation {
  ...
  src = ./source-dir;
}
```

However, if `source-dir` is a Subversion working copy, then all those annoying `.svn` subdirectories will also be copied to the store. Worse, the contents of those directories may change a lot, causing lots of spurious rebuilds. With `filterSource` you can filter out the `.svn` directories:

```
src = builtins.filterSource
  (path: type: type != "directory" || baseNameOf path !=
".svn")
  ./source-dir;
```

Thus, the first argument *e1* must be a predicate function that is called for each regular file, directory or symlink in the source tree *e2*. If the function returns `true`, the file is copied to the Nix store, otherwise it is omitted. The function is called with two arguments. The first is the full path of the file. The second is a string that identifies the type of the file, which is either `"regular"`, `"directory"`, `"symlink"` or `"unknown"` (for other kinds of files such as device nodes or fifos — but note that those cannot be copied to the Nix store, so if the predicate returns `true` for them, the copy will fail). If you exclude a directory, the entire corresponding subtree of *e2* will be excluded.

builtins.filter

f, list

Return a list consisting of the elements of *list* for which the

function *f* returns `true`.

builtins.fetchurl

url

Download the specified URL and return the path of the downloaded file. This function is not available if restricted evaluation mode is enabled.

builtins.fetchTarball

args

Download the specified URL, unpack it and return the path of the unpacked tree. The file must be a tape archive (`.tar`) compressed with `gzip`, `bzip2` or `xz`. The top-level path component of the files in the tarball is removed, so it is best if the tarball contains a single directory at top level. The typical use of the function is to obtain external Nix expression dependencies, such as a particular version of Nixpkgs, e.g.

```
with import (fetchTarball https://github.com/NixOS
/nixpkgs/archive/nixos-14.12.tar.gz) {};

stdenv.mkDerivation { … }
```

The fetched tarball is cached for a certain amount of time (1 hour by default) in `~/.cache/nix/tarballs/`. You can change the cache timeout either on the command line with `--tarball-ttl` *number-of-seconds* or in the Nix configuration file by adding the line `tarball-ttl =` *number-of-seconds*.

Note that when obtaining the hash with `nix-prefetch-url` the option `--unpack` is required.

This function can also verify the contents against a hash. In that

case, the function takes a set instead of a URL. The set requires the attribute `url` and the attribute `sha256`, e.g.

```
with import (fetchTarball {
  url = "https://github.com/NixOS/nixpkgs/archive/nixos-
14.12.tar.gz";
  sha256 =
"1jppksrfvbk5ypiqdz4cddxdl8z6zyzdb2srq8fcffr327ld5jj2";
}) {};

stdenv.mkDerivation { … }
```

This function is not available if restricted evaluation mode is enabled.

builtins.fetchGit

args

Fetch a path from git. *args* can be a URL, in which case the HEAD of the repo at that URL is fetched. Otherwise, it can be an attribute with the following attributes (all except `url` optional):

- url
  The URL of the repo.

- name
  The name of the directory the repo should be exported to in the store. Defaults to the basename of the URL.

- rev
  The git revision to fetch. Defaults to the tip of `ref`.

- ref
  The git ref to look for the requested revision under. This is often a branch or tag name. Defaults to `HEAD`.

By default, the `ref` value is prefixed with `refs/heads/`. As of Nix 2.3.0 Nix will not prefix `refs/heads/` if `ref` starts with `refs/`.

- submodules
  A Boolean parameter that specifies whether submodules should be checked out. Defaults to `false`.

- allRefs
  Whether to fetch all refs of the repository. With this argument being true, it's possible to load a `rev` from *any* `ref` (by default only `rev`s from the specified `ref` are supported).

Here are some examples of how to use `fetchGit`.

- To fetch a private repository over SSH:

```
builtins.fetchGit {
  url = "git@github.com:my-secret/repository.git";
  ref = "master";
  rev = "adab8b916a45068c044658c4158d81878f9ed1c3";
}
```

- To fetch an arbitrary reference:

```
builtins.fetchGit {
  url = "https://github.com/NixOS/nix.git";
  ref = "refs/heads/0.5-release";
}
```

- If the revision you're looking for is in the default branch of the git repository you don't strictly need to specify the branch name in the `ref` attribute.

  However, if the revision you're looking for is in a future branch for the non-default branch you will need to specify the the `ref`

attribute as well.

```
builtins.fetchGit {
  url = "https://github.com/nixos/nix.git";
  rev = "841fcbd04755c7a2865c51c1e2d3b045976b7452";
  ref = "1.11-maintenance";
}
```

> ***Note***
>
> *It is nice to always specify the branch which a revision belongs to. Without the branch being specified, the fetcher might fail if the default branch changes. Additionally, it can be confusing to try a commit from a non-default branch and see the fetch fail. If the branch is specified the fault is much more obvious.*

- If the revision you're looking for is in the default branch of the git repository you may omit the `ref` attribute.

```
builtins.fetchGit {
  url = "https://github.com/nixos/nix.git";
  rev = "841fcbd04755c7a2865c51c1e2d3b045976b7452";
}
```

- To fetch a specific tag:

```
builtins.fetchGit {
  url = "https://github.com/nixos/nix.git";
  ref = "refs/tags/1.9";
}
```

- To fetch the latest version of a remote branch:

```
builtins.fetchGit {
  url = "ssh://git@github.com/nixos/nix.git";
  ref = "master";
}
```

> ***Note***
>
> *Nix will refetch the branch in accordance with the option* `tarball-ttl`.

> ***Note***
>
> *This behavior is disabled in* Pure evaluation mode.

builtins.fetchClosure

args

Fetch a Nix store closure from a binary cache, rewriting it into content-addressed form. For example,

```
builtins.fetchClosure {
  fromStore = "https://cache.nixos.org";
  fromPath = /nix/store/r2jd6ygnmirm2g803mksqqjm4y39yi6i-
git-2.33.1;
  toPath = /nix/store/ldbhlwhh39wha58rm61bkiiwm6j7211j-
git-2.33.1;
}
```

fetches `/nix/store/r2jd...` from the specified binary cache, and rewrites it into the content-addressed store path `/nix/store/ldbh...`.

If `fromPath` is already content-addressed, or if you are allowing impure evaluation (`--impure`), then `toPath` may be omitted.

To find out the correct value for `toPath` given a `fromPath`, you can

use `nix store make-content-addressed`:

```
# nix store make-content-addressed --from
https://cache.nixos.org /nix/store
/r2jd6ygnmirm2g803mksqqjm4y39yi6i-git-2.33.1
rewrote '/nix/store/r2jd6ygnmirm2g803mksqqjm4y39yi6i-
git-2.33.1' to '/nix/store
/ldbhlwhh39wha58rm61bkiiwm6j7211j-git-2.33.1'
```

This function is similar to `builtins.storePath` in that it allows you to use a previously built store path in a Nix expression. However, it is more reproducible because it requires specifying a binary cache from which the path can be fetched. Also, requiring a content-addressed final store path avoids the need for users to configure binary cache public keys.

This function is only available if you enable the experimental feature `fetch-closure`.

builtins.elemAt

xs, *n*

Return element *n* from the list *xs*. Elements are counted starting from 0. A fatal error occurs if the index is out of bounds.

builtins.elem

x, *xs*

Return `true` if a value equal to *x* occurs in the list *xs*, and `false` otherwise.

builtins.div

e1, *e2*

Return the quotient of the numbers *e1* and *e2*.

builtins.dirOf

s

Return the directory part of the string *s*, that is, everything before the final slash in the string. This is similar to the GNU `dirname` command.

builtins.deepSeq

e1, e2

This is like `seq e1 e2`, except that *e1* is evaluated *deeply*: if it's a list or set, its elements or attributes are also evaluated recursively.

builtins.concatStringsSep

separator, list

Concatenate a list of strings with a separator between each element, e.g. `concatStringsSep "/" ["usr" "local" "bin"] == "usr/local/bin"`.

builtins.concatMap

f, list

This function is equivalent to `builtins.concatLists (map f list)` but is more efficient.

builtins.concatLists

lists

Concatenate a list of lists into a single list.

builtins.compareVersions

s1, s2

Compare two strings representing versions and return `-1` if version *s1* is older than version *s2*, `0` if they are the same, and `1` if *s1* is newer than *s2*. The version comparison algorithm is the same as the one used by `nix-env -u`.

builtins.ceil

double

Converts an IEEE-754 double-precision floating-point number (*double*) to the next higher integer.

If the datatype is neither an integer nor a “float”, an evaluation error will be thrown.

builtins.catAttrs

attr, list

Collect each attribute named *attr* from a list of attribute sets. Attrsets that don’t contain the named attribute are ignored. For example,

```
builtins.catAttrs "a" [{a = 1;} {b = 0;} {a = 2;}]
```

evaluates to `[1 2]`.

builtins.bitXor

e1, e2

Return the bitwise XOR of the integers *e1* and *e2*.

builtins.bitOr

e1, e2

Return the bitwise OR of the integers *e1* and *e2*.

builtins.bitAnd

e1, *e2*

Return the bitwise AND of the integers *e1* and *e2*.

builtins.baseNameOf

s

Return the *base name* of the string *s*, that is, everything following the final slash in the string. This is similar to the GNU `basename` command.

builtins.attrValues

set

Return the values of the attributes in the set *set* in the order corresponding to the sorted attribute names.

builtins.attrNames

set

Return the names of the attributes in the set *set* in an alphabetically sorted list. For instance, `builtins.attrNames { y = 1; x = "foo"; }` evaluates to `[ "x" "y" ]`.

builtins.any

pred, *list*

Return `true` if the function *pred* returns `true` for at least one element of *list*, and `false` otherwise.

builtins.all

pred, *list*

Return `true` if the function *pred* returns `true` for all elements of *list*, and `false` otherwise.

builtins.add

e1, *e2*

Return the sum of the numbers *e1* and *e2*.

builtins.abort

s

Abort Nix expression evaluation and print the error message *s*.

Nixpkgs Library Functions

Assert functions

lib.asserts.assertMsg

pred, *msg*

```
assertMsg :: Bool -> String -> Bool
```

Print a trace message if pred is false.

- `pred`

  Condition under which the *msg* should not be printed.

- `msg`

  Message to print.

Printing when the predicate is false:

```
assert lib.asserts.assertMsg ("foo" == "bar") "foo is not
bar, silly"
stderr> trace: foo is not bar, silly
stderr> assert failed
```

lib.asserts.assertOneOf

name, val, xs

```
assertOneOf :: String -> String -> StringList -> Bool
```

Specialized asserts.assertMsg for checking if val is one of the elements of xs. Useful for checking enums.

- `name`

  The name of the variable the user entered *val* into, for inclusion in the error message.

- `val`

  The value of what the user provided, to be compared against the values in *xs*.

- `xs`

  The list of valid values.

Ensuring a user provided a possible value:

```
let sslLibrary = "bearssl";
in lib.asserts.assertOneOf "sslLibrary" sslLibrary [
"openssl" "libressl" ];
=> false
stderr> trace: sslLibrary must be one of "openssl",
"libressl", but is: "bearssl"
```

Attribute-Set Functions

lib.attrset.attrByPath

attrPath, default, set

```
attrByPath :: [String] -> Any -> AttrSet -> Any
```

Return an attribute from within nested attribute sets.

- `attrPath`

  A list of strings representing the path through the nested attribute set *set*.

- `default`

  Default value if *attrPath* does not resolve to an existing value.

- `set`

  The nested attributeset to select values from.

Extracting a value from a nested attribute set:

```
let set = { a = { b = 3; }; };
in lib.attrsets.attrByPath [ "a" "b" ] 0 set
=> 3
```

No value at the path, instead using the default:

```
lib.attrsets.attrByPath [ "a" "b" ] 0 {}
=> 0
```

lib.attrsets.hasAttrByPath

attrPath, set

```
hasAttrByPath :: [String] -> AttrSet -> Bool
```

Determine if an attribute exists within a nested attribute set.

- `attrPath`

  A list of strings representing the path through the nested attribute set *set*.

- `set`

  The nested attributeset to check.

A nested value does exist inside a set:

```
lib.attrsets.hasAttrByPath
  [ "a" "b" "c" "d" ]
  { a = { b = { c = { d = 123; }; }; }; }
=> true
```

lib.attrsets.setAttrByPath
attrPath, *value*

`setAttrByPath :: [String] -> Any -> AttrSet`

Create a new attribute set with value set at the nested attribute location specified in attrPath.

- `attrPath`

  A list of strings representing the path through the nested attribute set.

- `value`

  The value to set at the location described by *attrPath*.

Creating a new nested attribute set:

```
lib.attrsets.setAttrByPath [ "a" "b" ] 3
=> { a = { b = 3; }; }
```

lib.attrsets.getAttrFromPath

attrPath, *set*

`getAttrFromPath :: [String] -> AttrSet -> Value`

Like except without a default, and it will throw if the value doesn't exist.

- `attrPath`

  A list of strings representing the path through the nested attribute set *set*.

- `set`

  The nested attribute set to find the value in.

Succesfully getting a value from an attribute set:

```
lib.attrsets.getAttrFromPath [ "a" "b" ] { a = { b = 3;
}; }
=> 3
```

Throwing after failing to get a value from an attribute set:

```
lib.attrsets.getAttrFromPath [ "x" "y" ] { }
=> error: cannot find attribute `x.y'
```

lib.attrsets.attrVals

nameList, *set*

`attrVals :: [String] -> AttrSet -> [Any]`

Return the specified attributes from a set. All values must exist.

- `nameList`

  The list of attributes to fetch from *set*. Each attribute name must exist on the attrbitue set.

- `set`

  The set to get attribute values from.

Getting several values from an attribute set:

```
lib.attrsets.attrVals [ "a" "b" "c" ] { a = 1; b = 2; c =
3; }
=> [ 1 2 3 ]
```

Getting missing values from an attribute set:

```
lib.attrsets.attrVals [ "d" ] { }
error: attribute 'd' missing
```

lib.attrsets.attrValues
attrs

`attrValues :: AttrSet -> [Any]`

Get all the attribute values from an attribute set.

- `attrs`

  The attribute set.

:

```
lib.attrsets.attrValues { a = 1; b = 2; c = 3; }
=> [ 1 2 3 ]
```

lib.attrsets.catAttrs

attr, *sets*

`catAttrs :: String -> [AttrSet] -> [Any]`

Collect each attribute named `attr' from the list of attribute sets, sets. Sets that don't contain the named attribute are ignored.

- `attr`

  Attribute name to select from each attribute set in *sets*.

- `sets`

  The list of attribute sets to select *attr* from.

Collect an attribute from a list of attribute sets.:

```
catAttrs "a" [{a = 1;} {b = 0;} {a = 2;}]
=> [ 1 2 ]
```

lib.attrsets.filterAttrs

pred, *name*, *value*, *set*, *name*, *value*

`filterAttrs :: (String -> Any -> Bool) -> AttrSet -> AttrSet`

Filter an attribute set by removing all attributes for which the given predicate return false.

- `pred`

  String -> Any -> Bool

- `name`

  The attribute's name

- `value`

  The attribute's value

- `set`

  The attribute set to filter

- `name`

  The attribute's name

- `value`

  The attribute's value

Filtering an attributeset:

```
filterAttrs (n: v: n == "foo") { foo = 1; bar = 2; }
=> { foo = 1; }
```

lib.attrsets.filterAttrsRecursive
pred, name, value, set, name, value

`filterAttrsRecursive :: (String -> Any -> Bool) -> AttrSet -> AttrSet`

Filter an attribute set recursively by removing all attributes for which the given predicate return false.

- `pred`

String -> Any -> Bool

- `name`

  The attribute's name

- `value`

  The attribute's value

- `set`

  The attribute set to filter

- `name`

  The attribute's name

- `value`

  The attribute's value

Recursively filtering an attribute set:

```
lib.attrsets.filterAttrsRecursive
  (n: v: v != null)
  {
    levelA = {
      example = "hi";
      levelB = {
        hello = "there";
        this-one-is-present = {
          this-is-excluded = null;
        };
      };
      this-one-is-also-excluded = null;
    };
    also-excluded = null;
  }
=> {
     levelA = {
       example = "hi";
       levelB = {
         hello = "there";
         this-one-is-present = { };
       };
     };
   }
```

lib.attrsets.foldAttrs

op, val, col, nul, list_of_attrs, val, col

`foldAttrs :: (Any -> Any -> Any) -> Any -> [AttrSets] -> Any`

Apply fold function to values grouped by key.

- `op`

  Any -> Any -> Any

- `val`

  An attribute's value

- `col`

  The result of previous op calls with other values and nul.

- `nul`

  The null-value, the starting value.

- `list_of_attrs`

  A list of attribute sets to fold together by key.

- `val`

  An attribute's value

- `col`

  The result of previous op calls with other values and nul.

Combining an attribute of lists in to one attribute set:

```
lib.attrsets.foldAttrs
  (n: a: [n] ++ a) []
  [
    { a = 2; b = 7; }
    { a = 3; }
    { b = 6; }
  ]
=> { a = [ 2 3 ]; b = [ 7 6 ]; }
```

lib.attrsets.collect

pred, *value*, *attrs*, *value*

`collect :: (Any -> Bool) -> AttrSet -> [Any]`

Recursively collect sets that verify a given predicate named pred from the set attrs. The recursion stops when pred returns true.

- `pred`

  Any -> Bool

- `value`

  The attribute set value.

- `attrs`

  The attribute set to recursively collect.

- `value`

  The attribute set value.

Collecting all lists from an attribute set:

```
lib.attrsets.collect isList { a = { b = ["b"]; }; c =
[1]; }
=> [["b"] [1]]
```

Collecting all attribute-sets which contain the outPath attribute name.:

```
collect (x: x ? outPath)
  { a = { outPath = "a/"; }; b = { outPath = "b/"; }; }
=> [{ outPath = "a/"; } { outPath = "b/"; }]
```

lib.attrsets.nameValuePair

name, value

`nameValuePair :: String -> Any -> AttrSet`

Utility function that creates a {name, value} pair as expected by builtins.listToAttrs.

- `name`

  The attribute name.

- `value`

  The attribute value.

Creating a name value pair:

```
nameValuePair "some" 6
=> { name = "some"; value = 6; }
```

lib.attrsets.mapAttrs

fn, name, value, name, value

``

Apply a function to each element in an attribute set, creating a new attribute set.

- `fn`

  String -> Any -> Any

- `name`

  The name of the attribute.

- `value`

  The attribute's value.

- `name`

  The name of the attribute.

- `value`

  The attribute's value.

Modifying each value of an attribute set:

```
lib.attrsets.mapAttrs
  (name: value: name + "-" + value)
  { x = "foo"; y = "bar"; }
=> { x = "x-foo"; y = "y-bar"; }
```

lib.attrsets.mapAttrs'
fn, name, value, set, name, value

`mapAttrs' :: (String -> Any -> { name = String; value = Any }) -> AttrSet -> AttrSet`

Like mapAttrs, but allows the name of each attribute to be changed in addition to the value. The applied function should return both the new name and value as a nameValuePair.

- `fn`

  String -> Any -> { name = String; value = Any }

- `name`

  The name of the attribute.

- `value`

  The attribute's value.

- `set`

  The attribute set to map over.

- `name`

  The name of the attribute.

- `value`

  The attribute's value.

Change the name and value of each attribute of an attribute set:

```
lib.attrsets.mapAttrs' (name: value:
lib.attrsets.nameValuePair ("foo_" + name) ("bar-" +
value))
   { x = "a"; y = "b"; }
=> { foo_x = "bar-a"; foo_y = "bar-b"; }
```

lib.attrsets.mapAttrsToList
fn, name, value, set, name, value

`mapAttrsToList :: (String -> Any -> Any) -> AttrSet -> [Any]`

Call fn for each attribute in the given set and return the result in a list.

- `fn`

  String -> Any -> Any

- `name`

  The name of the attribute.

- `value`

  The attribute's value.

- `set`

  The attribute set to map over.

- `name`

  The name of the attribute.

- `value`

  The attribute's value.

Combine attribute values and names in to a list:

```
lib.attrsets.mapAttrsToList (name: value:
"${name}=${value}")
   { x = "a"; y = "b"; }
=> [ "x=a" "y=b" ]
```

lib.attrsets.mapAttrsRecursive
f, name_path, value, set, name_path, value

`mapAttrsRecursive :: ([String] > Any -> Any) -> AttrSet -> AttrSet`

Like mapAttrs, except that it recursively applies itself to attribute sets. Also, the first argument of the argument function is a list of the names of the containing attributes.

- `f`

  [String] -> Any -> Any

- `name_path`

  The list of attribute names to this value.

- `value`

  The attribute's value.

- `set`

  The attribute set to recursively map over.

- `name_path`

  The list of attribute names to this value.

- `value`

  The attribute's value.

A contrived example of using lib.attrsets.mapAttrsRecursive:

```
mapAttrsRecursive
  (path: value: concatStringsSep "-" (path ++ [value]))
  {
    n = {
      a = "A";
      m = {
        b = "B";
        c = "C";
      };
    };
    d = "D";
  }
=> {
     n = {
       a = "n-a-A";
       m = {
         b = "n-m-b-B";
         c = "n-m-c-C";
       };
     };
     d = "d-D";
   }
```

lib.attrsets.mapAttrsRecursiveCond

cond, attributeset, f, name_path, value, set, attributeset, name_path, value

```
mapAttrsRecursiveCond :: (AttrSet -> Bool) -> ([ String ] -> Any -> Any) -> AttrSet -> AttrSet
```

Like mapAttrsRecursive, but it takes an additional predicate function that tells it whether to recursive into an attribute set. If it returns false, mapAttrsRecursiveCond does not recurse, but does apply the map function. It is returns true, it does recurse, and does not apply the map function.

- `cond`

  (AttrSet -> Bool)

- `attributeset`

  An attribute set.

- `f`

  [String] -> Any -> Any

- `name_path`

  The list of attribute names to this value.

- `value`

  The attribute's value.

- `set`

  The attribute set to recursively map over.

- `attributeset`

  An attribute set.

- `name_path`

  The list of attribute names to this value.

- `value`

  The attribute's value.

Only convert attribute values to JSON if the containing attribute set is marked for recursion:

```
lib.attrsets.mapAttrsRecursiveCond
  ({ recurse ? false, ... }: recurse)
  (name: value: builtins.toJSON value)
  {
    dorecur = {
      recurse = true;
      hello = "there";
    };
    dontrecur = {
      converted-to- = "json";
    };
  }
=> {
     dorecur = {
       hello = "\"there\"";
       recurse = "true";
     };
     dontrecur = "{\"converted-to\":\"json\"}";
   }
```

lib.attrsets.genAttrs

names, *f*, *name*, *name*

```
genAttrs :: [ String ] -> (String -> Any) -> AttrSet
```

Generate an attribute set by mapping a function over a list of attribute names.

- `names`

  Names of values in the resulting attribute set.

- `f`

String -> Any

- `name`

  The name of the attribute to generate a value for.

- `name`

  The name of the attribute to generate a value for.

Generate an attrset based on names only:

```
lib.attrsets.genAttrs [ "foo" "bar" ] (name: "x_${name}")
=> { foo = "x_foo"; bar = "x_bar"; }
```

lib.attrsets.isDerivation
value

`isDerivation :: Any -> Bool`

Check whether the argument is a derivation. Any set with { type = “derivation”; } counts as a derivation.

- `value`

  The value which is possibly a derivation.

A package is a derivation:

```
lib.attrsets.isDerivation (import <nixpkgs> {}).ruby
=> true
```

Anything else is not a derivation:

```
lib.attrsets.isDerivation "foobar"
=> false
```

lib.attrsets.toDerivation

path

```
toDerivation :: Path -> Derivation
```

Converts a store path to a fake derivation.

- `path`

  A store path to convert to a derivation.

lib.attrsets.optionalAttrs

cond, as

```
optionalAttrs :: Bool -> AttrSet
```

Conditionally return an attribute set or an empty attribute set.

- `cond`

  Condition under which the *as* attribute set is returned.

- `as`

  The attribute set to return if *cond* is true.

Return the provided attribute set when cond is true:

```
lib.attrsets.optionalAttrs true { my = "set"; }
=> { my = "set"; }
```

Return an empty attribute set when cond is false:

```
lib.attrsets.optionalAttrs false { my = "set"; }
=> { }
```

lib.attrsets.zipAttrsWithNames

names, *f*, *name*, *vs*, *sets*, *name*, *vs*

```
zipAttrsWithNames :: [ String ] -> (String -> [ Any ]
-> Any) -> [ AttrSet ] -> AttrSet
```

Merge sets of attributes and use the function f to merge attribute values where the attribute name is in names.

- `names`

  A list of attribute names to zip.

- `f`

  (String -> [Any] -> Any

- `name`

  The name of the attribute each value came from.

- `vs`

  A list of values collected from the list of attribute sets.

- `sets`

  A list of attribute sets to zip together.

- `name`

  The name of the attribute each value came from.

- `vs`

  A list of values collected from the list of attribute sets.

Summing a list of attribute sets of numbers:

```
lib.attrsets.zipAttrsWithNames
  [ "a" "b" ]
  (name: vals: "${name} ${toString (builtins.foldl' (a:
b: a + b) 0 vals)}")
  [
    { a = 1; b = 1; c = 1; }
    { a = 10; }
    { b = 100; }
    { c = 1000; }
  ]
=> { a = "a 11"; b = "b 101"; }
```

lib.attrsets.zipAttrsWith

f, *name*, *vs*, *sets*, *name*, *vs*

`zipAttrsWith :: (String -> [ Any ] -> Any) -> [ AttrSet ] -> AttrSet`

Merge sets of attributes and use the function f to merge attribute values. Similar to where all key names are passed for names.

- `f`

  (String -> [Any] -> Any

- `name`

  The name of the attribute each value came from.

- `vs`

  A list of values collected from the list of attribute sets.

- `sets`

A list of attribute sets to zip together.

- `name`

  The name of the attribute each value came from.

- `vs`

  A list of values collected from the list of attribute sets.

Summing a list of attribute sets of numbers:

```
lib.attrsets.zipAttrsWith
  (name: vals: "${name} ${toString (builtins.foldl' (a:
b: a + b) 0 vals)}")
  [
    { a = 1; b = 1; c = 1; }
    { a = 10; }
    { b = 100; }
    { c = 1000; }
  ]
=> { a = "a 11"; b = "b 101"; c = "c 1001"; }
```

lib.attrsets.zipAttrs

sets

`zipAttrs :: [ AttrSet ] -> AttrSet`

Merge sets of attributes and combine each attribute value in to a list. Similar to where the merge function returns a list of all values.

- `sets`

  A list of attribute sets to zip together.

Combining a list of attribute sets:

```
lib.attrsets.zipAttrs
  [
    { a = 1; b = 1; c = 1; }
    { a = 10; }
    { b = 100; }
    { c = 1000; }
  ]
=> { a = [ 1 10 ]; b = [ 1 100 ]; c = [ 1 1000 ]; }
```

lib.attrsets.recursiveUpdateUntil
pred, path, l, r, lhs, rhs, path, l, r

`recursiveUpdateUntil :: ( [ String ] -> AttrSet -> AttrSet -> Bool ) -> AttrSet -> AttrSet -> AttrSet`

Does the same as the update operator // except that attributes are merged until the given predicate is verified. The predicate should accept 3 arguments which are the path to reach the attribute, a part of the first attribute set and a part of the second attribute set. When the predicate is verified, the value of the first attribute set is replaced by the value of the second attribute set.

- `pred`

  [String] -> AttrSet -> AttrSet -> Bool

- `path`

  The path to the values in the left and right hand sides.

- `l`

  The left hand side value.

- `r`

The right hand side value.

- `lhs`

  The left hand attribute set of the merge.

- `rhs`

  The right hand attribute set of the merge.

- `path`

  The path to the values in the left and right hand sides.

- `l`

  The left hand side value.

- `r`

  The right hand side value.

Recursively merging two attribute sets:

```
lib.attrsets.recursiveUpdateUntil (path: l: r: path ==
["foo"])
  {
    # first attribute set
    foo.bar = 1;
    foo.baz = 2;
    bar = 3;
  }
  {
    #second attribute set
    foo.bar = 1;
    foo.quz = 2;
    baz = 4;
  }
=> {
  foo.bar = 1; # 'foo.*' from the second set
  foo.quz = 2; #
  bar = 3;     # 'bar' from the first set
  baz = 4;     # 'baz' from the second set
}
```

lib.attrsets.recursiveUpdate

lhs, rhs

`recursiveUpdate :: AttrSet -> AttrSet -> AttrSet`

A recursive variant of the update operator //. The recursion stops when one of the attribute values is not an attribute set, in which case the right hand side value takes precedence over the left hand side value.

- `lhs`

  The left hand attribute set of the merge.

- `rhs`

The right hand attribute set of the merge.

Recursively merging two attribute sets:

```
recursiveUpdate
  {
    boot.loader.grub.enable = true;
    boot.loader.grub.device = "/dev/hda";
  }
  {
    boot.loader.grub.device = "";
  }
=> {
  boot.loader.grub.enable = true;
  boot.loader.grub.device = "";
}
```

lib.attrsets.recurseIntoAttrs

attrs

`recurseIntoAttrs :: AttrSet -> AttrSet`

Make various Nix tools consider the contents of the resulting attribute set when looking for what to build, find, etc.

- `attrs`

  An attribute set to scan for derivations.

Making Nix look inside an attribute set:

```
{ pkgs ? import <nixpkgs> {} }:
{
  myTools = pkgs.lib.recurseIntoAttrs {
    inherit (pkgs) hello figlet;
  };
}
```

lib.attrsets.cartesianProductOfSets

set

```
cartesianProductOfSets :: AttrSet -> [ AttrSet ]
```

Return the cartesian product of attribute set value combinations.

- `set`

  An attribute set with attributes that carry lists of values.

Creating the cartesian product of a list of attribute values:

```
cartesianProductOfSets { a = [ 1 2 ]; b = [ 10 20 ]; }
=> [
     { a = 1; b = 10; }
     { a = 1; b = 20; }
     { a = 2; b = 10; }
     { a = 2; b = 20; }
   ]
```

Customisation functions

lib.customisation.overrideDerivation

drv, f

`overrideDerivation drv f' takes a derivation (i.e., the result of a call to the builtin function`derivation') and returns a new derivation in which the attributes of the original are overridden according to the function `f'. The function`f' is called

with the original derivation attributes.

- `drv`

  Function argument

- `f`

  Function argument

lib.customisation.makeOverridable
f, origArgs

`makeOverridable` takes a function from attribute set to attribute set and injects `override` attribute which can be used to override arguments of the function.

- `f`

  Function argument

- `origArgs`

  Function argument

lib.customisation.callPackageWith
autoArgs, fn, args

Call the package function in the file `fn' with the required arguments automatically. The function is called with the arguments`args', but any missing arguments are obtained from `autoArgs'. This function is intended to be partially parameterised, e.g.,

- `autoArgs`

Function argument

- `fn`

  Function argument

- `args`

  Function argument

lib.customisation.callPackagesWith
autoArgs, fn, args

Like callPackage, but for a function that returns an attribute set of derivations. The override function is added to the individual attributes.

- `autoArgs`

  Function argument

- `fn`

  Function argument

- `args`

  Function argument

lib.customisation.extendDerivation
condition, passthru, drv

Add attributes to each output of a derivation without changing the derivation itself and check a given condition when evaluating.

- `condition`

Function argument

- `passthru`

  Function argument

- `drv`

  Function argument

lib.customisation.hydraJob
drv

Strip a derivation of all non-essential attributes, returning only those needed by hydra-eval-jobs. Also strictly evaluate the result to ensure that there are no thunks kept alive to prevent garbage collection.

- `drv`

  Function argument

lib.customisation.makeScope
newScope, f

Make a set of packages with a common scope. All packages called with the provided `callPackage' will be evaluated with the same arguments. Any package in the set may depend on any other. The`overrideScope'`function allows subsequent modification of the package set in a consistent way, i.e. all packages in the set will be called with the overridden packages. The package sets may be hierarchical: the packages in the set are called with the scope provided by`newScope' and the set provides a `newScope' attribute which can form the parent scope for later package sets.

- `newScope`

  Function argument

- `f`

  Function argument

lib.customisation.makeScopeWithSplicing
splicePackages, newScope, otherSplices, keep, extra, f

Like the above, but aims to support cross compilation. It's still ugly, but hopefully it helps a little bit.

- `splicePackages`

  Function argument

- `newScope`

  Function argument

- `otherSplices`

  Function argument

- `keep`

  Function argument

- `extra`

  Function argument

- `f`

Function argument

Debugging functions

lib.debug.traceIf

pred, msg, x

```
traceIf :: bool -> string -> a -> a
```

Conditionally trace the supplied message, based on a predicate.

- `pred`

  Predicate to check

- `msg`

  Message that should be traced

- `x`

  Value to return

lib.debug.traceIf usage example:

```
traceIf true "hello" 3
trace: hello
=> 3
```

lib.debug.traceValFn

f, x

```
traceValFn :: (a -> b) -> a -> a
```

Trace the supplied value after applying a function to it, and return the original value.

- `f`

  Function to apply

- `x`

  Value to trace and return

lib.debug.traceValFn usage example:

```
traceValFn (v: "mystring ${v}") "foo"
trace: mystring foo
=> "foo"
```

lib.debug.traceVal

—

`traceVal :: a -> a`

Trace the supplied value and return it.

lib.debug.traceVal usage example:

```
traceVal 42
# trace: 42
=> 42
```

lib.debug.traceSeq

x, *y*

`traceSeq :: a -> b -> b`

`builtins.trace`, but the value is `builtins.deepSeq`ed first.

- `x`

The value to trace

- `y`

  The value to return

lib.debug.traceSeq usage example:

```
trace { a.b.c = 3; } null
trace: { a = <CODE>; }
=> null
traceSeq { a.b.c = 3; } null
trace: { a = { b = { c = 3; }; }; }
=> null
```

lib.debug.traceSeqN
depth, x, y

Like `traceSeq`, but only evaluate down to depth n. This is very useful because lots of `traceSeq` usages lead to an infinite recursion.

- `depth`

  Function argument

- `x`

  Function argument

- `y`

  Function argument

lib.debug.traceSeqN usage example:

```
traceSeqN 2 { a.b.c = 3; } null
trace: { a = { b = {…}; }; }
=> null
```

lib.debug.traceValSeqFn
f, v

A combination of `traceVal` and `traceSeq` that applies a provided function to the value to be traced after `deepSeq`ing it.

- `f`

  Function to apply

- `v`

  Value to trace

lib.debug.traceValSeq
—

A combination of `traceVal` and `traceSeq`.

lib.debug.traceValSeqNFn
f, depth, v

A combination of `traceVal` and `traceSeqN` that applies a provided function to the value to be traced.

- `f`

  Function to apply

- `depth`

  Function argument

- `v`

  Value to trace

lib.debug.traceValSeqN

—

A combination of `traceVal` and `traceSeqN`.

lib.debug.traceFnSeqN
depth, name, f, v

Trace the input and output of a function `f` named `name`, both down to `depth`.

- `depth`

  Function argument

- `name`

  Function argument

- `f`

  Function argument

- `v`

  Function argument

lib.debug.traceFnSeqN usage example:

```
traceFnSeqN 2 "id" (x: x) { a.b.c = 3; }
trace: { fn = "id"; from = { a.b = {…}; }; to = { a.b =
{…}; }; }
=> { a.b.c = 3; }
```

lib.debug.runTests
tests

Evaluate a set of tests. A test is an attribute set `{expr, expected}`, denoting an expression and its expected result. The result is a list of failed tests, each represented as `{name, expected, actual}`, denoting the attribute name of the failing test and its expected and actual results.

- `tests`

  Tests to run

lib.debug.testAllTrue
expr

Create a test assuming that list elements are `true`.

- `expr`

  Function argument

lib.debug.testAllTrue usage example:

```
{ testX = allTrue [ true ]; }
```

Generator functions

lib.generators.mkValueStringDefault
v

Convert a value to a sensible default string representation. * The

builtin `toString` function has some strange defaults, * suitable for bash scripts but not much else.

- `v`

  Function argument

lib.generators.mkKeyValueDefault
pattern, mkValueString, sep, k, v, mkValueString

Generate a line of key k and value v, separated by * character sep. If sep appears in k, it is escaped. * Helper for synaxes with different separators. mkValueString specifies how values should be formatted. mkKeyValueDefault {} ":" "f:oo" "bar" * > "f:oo:bar"

- `pattern`

  Structured function argument

- `mkValueString`

  Function argument

- `sep`

  Function argument

- `k`

  Function argument

- `v`

  Function argument

- `mkValueString`

Function argument

lib.generators.toKeyValue
pattern, mkKeyValue, listsAsDuplicateKeys, mkKeyValue, listsAsDuplicateKeys

Generate a key-value-style config file from an attrset. mkKeyValue is the same as in toINI.

- `pattern`

  Structured function argument

- `mkKeyValue`

  Function argument

- `listsAsDuplicateKeys`

  Function argument

- `mkKeyValue`

  Function argument

- `listsAsDuplicateKeys`

  Function argument

lib.generators.toINI
pattern, mkSectionName, mkKeyValue, listsAsDuplicateKeys, attrsOfAttrs, mkSectionName, mkKeyValue, listsAsDuplicateKeys

Generate an INI-style config file from an * attrset of sections to an attrset of key-value pairs. generators.toINI {} { * foo = { hi = "${pkgs.hello}"; ciao = "bar"; }; * baz = { "also, integers" = 42; }; * } >

[baz] *> also, integers=42 > > [foo]* > ciao=bar *> hi=/nix/store /y93qql1p5ggfnaqjjqhxcw0vqw95rlz0-hello-2.10* * The mk* configuration attributes can generically change * the way sections and key-value strings are generated. For more examples see the test cases in ./tests/misc.nix.

- `pattern`

  Structured function argument

- `mkSectionName`

  apply transformations (e.g. escapes) to section names

- `mkKeyValue`

  format a setting line from key and value

- `listsAsDuplicateKeys`

  allow lists as values for duplicate keys

- `attrsOfAttrs`

  Function argument

- `mkSectionName`

  apply transformations (e.g. escapes) to section names

- `mkKeyValue`

  format a setting line from key and value

- `listsAsDuplicateKeys`

allow lists as values for duplicate keys

lib.generators.toINIWithGlobalSection
pattern, mkSectionName, mkKeyValue, listsAsDuplicateKeys, pattern, globalSection, sections, mkSectionName, mkKeyValue, listsAsDuplicateKeys, globalSection, sections

Generate an INI-style config file from an attrset * specifying the global section (no header), and an * attrset of sections to an attrset of key-value pairs. generators.toINIWithGlobalSection {} { * globalSection = { * someGlobalKey = "hi"; * }; * sections = { * foo = { hi = "${pkgs.hello}"; ciao = "bar"; }; * baz = { "also, integers" = 42; }; * } > someGlobalKey=hi *> > [baz] > also, integers=42 > > [foo] >* ciao=bar *> hi=/nix/store/y93qql1p5ggfnaqjjqhxcw0vqw95rlz0-hello-2.10 * The mk* configuration attributes can generically change * the way sections and key-value strings are generated. For more examples see the test cases in ./tests/misc.nix. If you don't need a global section, you can also use * `generators.toINI` directly, which only takes * the part in `sections`.

- `pattern`

  Structured function argument

- `mkSectionName`

  apply transformations (e.g. escapes) to section names

- `mkKeyValue`

  format a setting line from key and value

- `listsAsDuplicateKeys`

  allow lists as values for duplicate keys

- `pattern`

  Structured function argument

- `globalSection`

  Function argument

- `sections`

  Function argument

- `mkSectionName`

  apply transformations (e.g. escapes) to section names

- `mkKeyValue`

  format a setting line from key and value

- `listsAsDuplicateKeys`

  allow lists as values for duplicate keys

- `globalSection`

  Function argument

- `sections`

  Function argument

lib.generators.toGitINI
attrs

Generate a git-config file from an attrset. It has two major

differences from the regular INI format: 1. values are indented with tabs * 2. sections can have sub-sections generators.toGitINI { * url."ssh://git@github.com/".insteadOf ="https://github.com"; * user.name = "edolstra"; * } > [url "ssh://git@github.com/"] > *insteadOf = https://github.com/* > > *[user]* > name = edolstra

- `attrs`

  Function argument

lib.generators.toJSON

—

Generates JSON from an arbitrary (non-function) value. * For more information see the documentation of the builtin.

lib.generators.toYAML

—

YAML has been a strict superset of JSON since 1.2, so we * use toJSON. Before it only had a few differences referring * to implicit typing rules, so it should work with older * parsers as well.

lib.generators.toPretty
pattern, allowPrettyValues, multiline, allowPrettyValues, multiline

Pretty print a value, akin to `builtins.trace`. * Should probably be a builtin as well.

- `pattern`

  Structured function argument

- `allowPrettyValues`

  If this option is true, attrsets like { __pretty = fn; val = …; } will

use fn to convert val to a pretty printed representation. (This means fn is type Val -> String.)

- `multiline`

  If this option is true, the output is indented with newlines for attribute sets and lists

- `allowPrettyValues`

  If this option is true, attrsets like { __pretty = fn; val = …; } will use fn to convert val to a pretty printed representation. (This means fn is type Val -> String.)

- `multiline`

  If this option is true, the output is indented with newlines for attribute sets and lists

lib.generators.toDhall
v

Translate a simple Nix expression to Dhall notation. * Note that integers are translated to Integer and never * the Natural type.

- `v`

  Function argument

List manipulation functions

lib.lists.singleton
x

`singleton :: a -> [a]`

Create a list consisting of a single element. `singleton x` is

sometimes more convenient with respect to indentation than `[x]` when x spans multiple lines.

- `x`

  Function argument

lib.lists.singleton usage example:

```
singleton "foo"
=> [ "foo" ]
```

lib.lists.forEach

xs, f

`forEach :: [a] -> (a -> b) -> [b]`

Apply the function to each element in the list. Same as `map`, but arguments flipped.

- `xs`

  Function argument

- `f`

  Function argument

lib.lists.forEach usage example:

```
forEach [ 1 2 ] (x:
toString x
)
=> [ "1" "2" ]
```

lib.lists.foldr

op, nul, list

`foldr :: (a -> b -> b) -> b -> [a] -> b`

"right fold" a binary function `op` between successive elements of `list` with `nul` as the starting value, i.e., `foldr op nul [x_1 x_2 ... x_n] == op x_1 (op x_2 ... (op x_n nul))`.

- `op`

  Function argument

- `nul`

  Function argument

- `list`

  Function argument

lib.lists.foldr usage example:

```
concat = foldr (a: b: a + b) "z"
concat [ "a" "b" "c" ]
=> "abcz"
# different types
strange = foldr (int: str: toString (int + 1) + str) "a"
strange [ 1 2 3 4 ]
=> "2345a"
```

lib.lists.fold

—

`fold` is an alias of `foldr` for historic reasons

lib.lists.foldl

op, nul, list

`foldl :: (b -> a -> b) -> b -> [a] -> b`

"left fold", like `foldr`, but from the left: `foldl op nul [x_1 x_2 ... x_n] == op (... (op (op nul x_1) x_2) ... x_n)`.

- `op`

  Function argument

- `nul`

  Function argument

- `list`

  Function argument

lib.lists.foldl usage example:

```
lconcat = foldl (a: b: a + b) "z"
lconcat [ "a" "b" "c" ]
=> "zabc"
# different types
lstrange = foldl (str: int: str + toString (int + 1)) "a"
lstrange [ 1 2 3 4 ]
=> "a2345"
```

lib.lists.foldl'

—

`foldl' :: (b -> a -> b) -> b -> [a] -> b`

Strict version of `foldl`.

lib.lists.imap0

f, list

```
imap0 :: (int -> a -> b) -> [a] -> [b]
```

Map with index starting from 0

- `f`

  Function argument

- `list`

  Function argument

lib.lists.imap0 usage example:

```
imap0 (i: v: "${v}-${toString i}") ["a" "b"]
=> [ "a-0" "b-1" ]
```

lib.lists.imap1

f, list

```
imap1 :: (int -> a -> b) -> [a] -> [b]
```

Map with index starting from 1

- `f`

  Function argument

- `list`

  Function argument

lib.lists.imap1 usage example:

```
imap1 (i: v: "${v}-${toString i}") ["a" "b"]
=> [ "a-1" "b-2" ]
```

lib.lists.concatMap

—

`concatMap :: (a -> [b]) -> [a] -> [b]`

Map and concatenate the result.

lib.lists.concatMap usage example:

```
concatMap (x: [x] ++ ["z"]) ["a" "b"]
=> [ "a" "z" "b" "z" ]
```

lib.lists.flatten

x

Flatten the argument into a single list; that is, nested lists are spliced into the top-level lists.

- `x`

  Function argument

lib.lists.flatten usage example:

```
flatten [1 [2 [3] 4] 5]
=> [1 2 3 4 5]
flatten 1
=> [1]
```

lib.lists.remove

e

`remove :: a -> [a] -> [a]`

Remove elements equal to ‘e’ from a list. Useful for buildInputs.

- `e`

  Element to remove from the list

lib.lists.remove usage example:

```
remove 3 [ 1 3 4 3 ]
=> [ 1 4 ]
```

lib.lists.findSingle
pred, default, multiple, list

```
findSingle :: (a -> bool) -> a -> a -> [a] -> a
```

Find the sole element in the list matching the specified predicate, returns `default` if no such element exists, or `multiple` if there are multiple matching elements.

- `pred`

  Predicate

- `default`

  Default value to return if element was not found.

- `multiple`

  Default value to return if more than one element was found

- `list`

  Input list

lib.lists.findSingle usage example:

```
findSingle (x: x == 3) "none" "multiple" [ 1 3 3 ]
=> "multiple"
findSingle (x: x == 3) "none" "multiple" [ 1 3 ]
=> 3
findSingle (x: x == 3) "none" "multiple" [ 1 9 ]
=> "none"
```

lib.lists.findFirst
pred, default, list

`findFirst :: (a -> bool) -> a -> [a] -> a`

Find the first element in the list matching the specified predicate or return `default` if no such element exists.

- `pred`

  Predicate

- `default`

  Default value to return

- `list`

  Input list

lib.lists.findFirst usage example:

```
findFirst (x: x > 3) 7 [ 1 6 4 ]
=> 6
findFirst (x: x > 9) 7 [ 1 6 4 ]
=> 7
```

lib.lists.any

—

```
any :: (a -> bool) -> [a] -> bool
```

Return true if function `pred` returns true for at least one element of `list`.

lib.lists.any usage example:

```
any isString [ 1 "a" { } ]
=> true
any isString [ 1 { } ]
=> false
```

lib.lists.all

—

```
all :: (a -> bool) -> [a] -> bool
```

Return true if function `pred` returns true for all elements of `list`.

lib.lists.all usage example:

```
all (x: x < 3) [ 1 2 ]
=> true
all (x: x < 3) [ 1 2 3 ]
=> false
```

lib.lists.count

pred

```
count :: (a -> bool) -> [a] -> int
```

Count how many elements of `list` match the supplied predicate function.

- `pred`

  Predicate

lib.lists.count usage example:

```
count (x: x == 3) [ 3 2 3 4 6 ]
=> 2
```

lib.lists.optional
cond, elem

`optional :: bool -> a -> [a]`

Return a singleton list or an empty list, depending on a boolean value. Useful when building lists with optional elements (e.g. `++ optional (system == "i686-linux") firefox').

- `cond`

  Function argument

- `elem`

  Function argument

lib.lists.optional usage example:

```
optional true "foo"
=> [ "foo" ]
optional false "foo"
=> [ ]
```

lib.lists.optionals
cond, elems

`optionals :: bool -> [a] -> [a]`

Return a list or an empty list, depending on a boolean value.

- `cond`

  Condition

- `elems`

  List to return if condition is true

lib.lists.optionals usage example:

```
optionals true [ 2 3 ]
=> [ 2 3 ]
optionals false [ 2 3 ]
=> [ ]
```

lib.lists.toList

x

If argument is a list, return it; else, wrap it in a singleton list. If you're using this, you should almost certainly reconsider if there isn't a more "well-typed" approach.

- `x`

  Function argument

lib.lists.toList usage example:

```
toList [ 1 2 ]
=> [ 1 2 ]
toList "hi"
=> [ "hi "]
```

lib.lists.range

first, last

```
range :: int -> int -> [int]
```

Return a list of integers from `first' up to and including`last'.

- `first`

  First integer in the range

- `last`

  Last integer in the range

lib.lists.range usage example:

```
range 2 4
=> [ 2 3 4 ]
range 3 2
=> [ ]
```

lib.lists.partition

—

```
(a -> bool) -> [a] -> { right :: [a], wrong :: [a] }
```

Splits the elements of a list in two lists, `right` and `wrong`, depending on the evaluation of a predicate.

lib.lists.partition usage example:

```
partition (x: x > 2) [ 5 1 2 3 4 ]
=> { right = [ 5 3 4 ]; wrong = [ 1 2 ]; }
```

lib.lists.groupBy'

op, nul, pred, lst

Splits the elements of a list into many lists, using the return value of a predicate. Predicate should return a string which becomes keys of attrset `groupBy' returns.

- `op`

  Function argument

- `nul`

  Function argument

- `pred`

  Function argument

- `lst`

  Function argument

lib.lists.groupBy' usage example:

```
groupBy (x: boolToString (x > 2)) [ 5 1 2 3 4 ]
=> { true = [ 5 3 4 ]; false = [ 1 2 ]; }
groupBy (x: x.name) [ {name = "icewm"; script = "icewm
&";}
{name = "xfce";  script = "xfce4-session &";}
{name = "icewm"; script = "icewmbg &";}
{name = "mate";  script = "gnome-session &";}
]
=> { icewm = [ { name = "icewm"; script = "icewm &"; }
{ name = "icewm"; script = "icewmbg &"; } ];
mate  = [ { name = "mate";  script = "gnome-session &"; }
];
xfce  = [ { name = "xfce";  script = "xfce4-session &"; }
];
}

groupBy' builtins.add 0 (x: boolToString (x > 2)) [ 5 1 2
3 4 ]
=> { true = 12; false = 3; }
```

lib.lists.zipListsWith

f, fst, snd

`zipListsWith :: (a -> b -> c) -> [a] -> [b] -> [c]`

Merges two lists of the same size together. If the sizes aren't the same the merging stops at the shortest. How both lists are merged is defined by the first argument.

- `f`

  Function to zip elements of both lists

- `fst`

  First list

- `snd`

  Second list

lib.lists.zipListsWith usage example:

```
zipListsWith (a: b: a + b) ["h" "l"] ["e" "o"]
=> ["he" "lo"]
```

lib.lists.zipLists

—

`zipLists :: [a] -> [b] -> [{ fst :: a, snd :: b}]`

Merges two lists of the same size together. If the sizes aren't the same the merging stops at the shortest.

lib.lists.zipLists usage example:

```
zipLists [ 1 2 ] [ "a" "b" ]
=> [ { fst = 1; snd = "a"; } { fst = 2; snd = "b"; } ]
```

lib.lists.reverseList

xs

`reverseList :: [a] -> [a]`

Reverse the order of the elements of a list.

- `xs`

  Function argument

lib.lists.reverseList usage example:

```
reverseList [ "b" "o" "j" ]
=> [ "j" "o" "b" ]
```

lib.lists.listDfs

stopOnCycles, *before*, *list*

Depth-First Search (DFS) for lists `list != []`.

- `stopOnCycles`

  Function argument

- `before`

  Function argument

- `list`

  Function argument

lib.lists.listDfs usage example:

```
listDfs true hasPrefix [ "/home/user" "other" "/" "/home"
]
== { minimal = "/";                  # minimal element
visited = [ "/home/user" ];     # seen elements (in
reverse order)
rest    = [ "/home" "other" ];  # everything else
}

listDfs true hasPrefix [ "/home/user" "other" "/" "/home"
"/" ]
== { cycle   = "/";                  # cycle encountered
at this element
loops   = [ "/" ];              # and continues to these
elements
visited = [ "/" "/home/user" ]; # elements leading to the
cycle (in reverse order)
rest    = [ "/home" "other" ];  # everything else
```

lib.lists.toposort
before, list

Sort a list based on a partial ordering using DFS. This implementation is O(N^2), if your ordering is linear, use `sort` instead.

- `before`

  Function argument

- `list`

  Function argument

lib.lists.toposort usage example:

```
toposort hasPrefix [ "/home/user" "other" "/" "/home" ]
== { result = [ "/" "/home" "/home/user" "other" ]; }

toposort hasPrefix [ "/home/user" "other" "/" "/home" "/"
]
== { cycle = [ "/home/user" "/" "/" ]; # path leading to
a cycle
loops = [ "/" ]; }                # loops back to these
elements

toposort hasPrefix [ "other" "/home/user" "/home" "/" ]
== { result = [ "other" "/" "/home" "/home/user" ]; }

toposort (a: b: a < b) [ 3 2 1 ] == { result = [ 1 2 3 ];
}
```

lib.lists.sort

—

Sort a list based on a comparator function which compares two elements and returns true if the first argument is strictly below the second argument. The returned list is sorted in an increasing order. The implementation does a quick-sort.

lib.lists.sort usage example:

```
sort (a: b: a < b) [ 5 3 7 ]
=> [ 3 5 7 ]
```

lib.lists.compareLists

cmp, a, b

Compare two lists element-by-element.

- `cmp`

Function argument

- `a`

  Function argument

- `b`

  Function argument

lib.lists.compareLists usage example:

```
compareLists compare [] []
=> 0
compareLists compare [] [ "a" ]
=> -1
compareLists compare [ "a" ] []
=> 1
compareLists compare [ "a" "b" ] [ "a" "c" ]
=> -1
```

lib.lists.naturalSort
lst

Sort list using “Natural sorting”. Numeric portions of strings are sorted in numeric order.

- `lst`

  Function argument

lib.lists.naturalSort usage example:

```
naturalSort ["disk11" "disk8" "disk100" "disk9"]
=> ["disk8" "disk9" "disk11" "disk100"]
naturalSort ["10.46.133.149" "10.5.16.62" "10.54.16.25"]
=> ["10.5.16.62" "10.46.133.149" "10.54.16.25"]
naturalSort ["v0.2" "v0.15" "v0.0.9"]
=> [ "v0.0.9" "v0.2" "v0.15" ]
```

lib.lists.take

count

```
take :: int -> [a] -> [a]
```

Return the first (at most) N elements of a list.

- `count`

  Number of elements to take

lib.lists.take usage example:

```
take 2 [ "a" "b" "c" "d" ]
=> [ "a" "b" ]
take 2 [ ]
=> [ ]
```

lib.lists.drop

count, list

```
drop :: int -> [a] -> [a]
```

Remove the first (at most) N elements of a list.

- `count`

  Number of elements to drop

- `list`

  Input list

lib.lists.drop usage example:

```
drop 2 [ "a" "b" "c" "d" ]
=> [ "c" "d" ]
drop 2 [ ]
=> [ ]
```

lib.lists.sublist
start, count, list

`sublist :: int -> int -> [a] -> [a]`

Return a list consisting of at most `count` elements of `list`, starting at index `start`.

- `start`

  Index at which to start the sublist

- `count`

  Number of elements to take

- `list`

  Input list

lib.lists.sublist usage example:

```
sublist 1 3 [ "a" "b" "c" "d" "e" ]
=> [ "b" "c" "d" ]
sublist 1 3 [ ]
=> [ ]
```

lib.lists.last

list

`last :: [a] -> a`

Return the last element of a list.

- `list`

  Function argument

lib.lists.last usage example:

```
last [ 1 2 3 ]
=> 3
```

lib.lists.init

list

`init :: [a] -> [a]`

Return all elements but the last.

- `list`

  Function argument

lib.lists.init usage example:

```
init [ 1 2 3 ]
=> [ 1 2 ]
```

lib.lists.crossLists

—

Return the image of the cross product of some lists by a function.

lib.lists.crossLists usage example:

```
crossLists (x:y: "${toString x}${toString y}") [[1 2] [3
4]]
=> [ "13" "14" "23" "24" ]
```

lib.lists.unique

—

`unique :: [a] -> [a]`

Remove duplicate elements from the list. O(n^2) complexity.

lib.lists.unique usage example:

```
unique [ 3 2 3 4 ]
=> [ 3 2 4 ]
```

lib.lists.intersectLists

e

Intersects list 'e' and another list. O(nm) complexity.

- `e`

  Function argument

lib.lists.intersectLists usage example:

```
intersectLists [ 1 2 3 ] [ 6 3 2 ]
=> [ 3 2 ]
```

lib.lists.subtractLists
e

Subtracts list ‘e’ from another list. O(nm) complexity.

- `e`

  Function argument

lib.lists.subtractLists usage example:

```
subtractLists [ 3 2 ] [ 1 2 3 4 5 3 ]
=> [ 1 4 5 ]
```

lib.lists.mutuallyExclusive
a, b

Test if two lists have no common element. It should be slightly more efficient than (intersectLists a b == [])

- `a`

  Function argument

- `b`

  Function argument

Meta functions

lib.meta.addMetaAttrs
newAttrs, drv

Add to or override the meta attributes of the given derivation.

- `newAttrs`

Function argument

- `drv`

  Function argument

lib.meta.addMetaAttrs usage example:

```
addMetaAttrs {description = "Bla blah";} somePkg
```

lib.meta.dontDistribute
drv

Disable Hydra builds of given derivation.

- `drv`

  Function argument

lib.meta.setName
name, drv

Change the symbolic name of a package for presentation purposes
(i.e., so that nix-env users can tell them apart).

- `name`

  Function argument

- `drv`

  Function argument

lib.meta.updateName
updater, drv

Like `setName', but takes the previous name as an argument.

- `updater`

  Function argument

- `drv`

  Function argument

lib.meta.updateName usage example:

```
updateName (oldName: oldName + "-experimental") somePkg
```

lib.meta.appendToName
suffix

Append a suffix to the name of a package (before the version part).

- `suffix`

  Function argument

lib.meta.mapDerivationAttrset
f, set

Apply a function to each derivation and only to derivations in an attrset.

- `f`

  Function argument

- `set`

  Function argument

lib.meta.setPrio
priority

Set the nix-env priority of the package.

- `priority`

  Function argument

lib.meta.lowPrio

—

Decrease the nix-env priority of the package, i.e., other versions/variants of the package will be preferred.

lib.meta.lowPrioSet
set

Apply lowPrio to an attrset with derivations

- `set`

  Function argument

lib.meta.hiPrio

—

Increase the nix-env priority of the package, i.e., this version/variant of the package will be preferred.

lib.meta.hiPrioSet
set

Apply hiPrio to an attrset with derivations

- `set`

Function argument

lib.meta.platformMatch
platform, elem

Check to see if a platform is matched by the given `meta.platforms` element.

- `platform`

  Function argument

- `elem`

  Function argument

lib.meta.availableOn
platform, pkg

Check if a package is available on a given platform.

- `platform`

  Function argument

- `pkg`

  Function argument

lib.meta.getLicenseFromSpdxId

—

`getLicenseFromSpdxId :: str -> AttrSet`

Get the corresponding attribute in lib.licenses from the SPDX ID.
For SPDX IDs, see https://spdx.org/licenses

lib.meta.getLicenseFromSpdxId usage example:

```
lib.getLicenseFromSpdxId "MIT" == lib.licenses.mit
=> true
lib.getLicenseFromSpdxId "mIt" == lib.licenses.mit
=> true
lib.getLicenseFromSpdxId "MY LICENSE"
=> trace: warning: getLicenseFromSpdxId: No license
matches the given SPDX ID: MY LICENSE
=> { shortName = "MY LICENSE"; }
```

lib.meta.getExe

x

```
getExe :: derivation -> string
```

Get the path to the main program of a derivation with either meta.mainProgram or pname or name

- x

  Function argument

lib.meta.getExe usage example:

```
getExe pkgs.hello
=> "/nix/store/g124820p9hlv4lj8qplzxw1c44dxaw1k-hello-
2.12/bin/hello"
getExe pkgs.mustache-go
=> "/nix/store/am9ml4f4ywvivxnkiaqwr0hyxka1xjsf-mustache-
go-1.3.0/bin/mustache"
```

Modules functions

lib.modules.evalModules

—

Evaluate a set of modules. The result is a set with the attributes:

lib.modules.collectStructuredModules

—

Collects all modules recursively into the form

lib.modules.setDefaultModuleLocation
file, m

Wrap a module with a default location for reporting errors.

- `file`

  Function argument

- `m`

  Function argument

lib.modules.unifyModuleSyntax
file, key, m

Massage a module into canonical form, that is, a set consisting of ‘options’, ‘config’ and ‘imports’ attributes.

- `file`

  Function argument

- `key`

  Function argument

- `m`

Function argument

lib.modules.mergeModules
prefix, modules

Merge a list of modules. This will recurse over the option declarations in all modules, combining them into a single set. At the same time, for each option declaration, it will merge the corresponding option definitions in all machines, returning them in the 'value' attribute of each option.

- `prefix`

  Function argument

- `modules`

  Function argument

lib.modules.byName
attr, f, modules

byName is like foldAttrs, but will look for attributes to merge in the specified attribute name.

- `attr`

  Function argument

- `f`

  Function argument

- `modules`

  Function argument

lib.modules.mergeOptionDecls

—

Merge multiple option declarations into a single declaration. In general, there should be only one declaration of each option. The exception is the ‘options’ attribute, which specifies sub-options. These can be specified multiple times to allow one module to add sub-options to an option declared somewhere else (e.g. multiple modules define sub-options for ‘fileSystems’).

lib.modules.evalOptionValue
loc, opt, defs

Merge all the definitions of an option to produce the final config value.

- `loc`

  Function argument

- `opt`

  Function argument

- `defs`

  Function argument

lib.modules.pushDownProperties
cfg

Given a config set, expand mkMerge properties, and push down the other properties into the children. The result is a list of config sets that do not have properties at top-level. For example,

- `cfg`

Function argument

lib.modules.dischargeProperties
def

Given a config value, expand mkMerge properties, and discharge any mkIf conditions. That is, this is the place where mkIf conditions are actually evaluated. The result is a list of config values. For example, ‘mkIf false x’ yields ‘[]’, ‘mkIf true x’ yields ‘[x]’, and

- `def`

  Function argument

lib.modules.filterOverrides
defs

Given a list of config values, process the mkOverride properties, that is, return the values that have the highest (that is, numerically lowest) priority, and strip the mkOverride properties. For example,

- `defs`

  Function argument

lib.modules.sortProperties
defs

Sort a list of properties. The sort priority of a property is 1000 by default, but can be overridden by wrapping the property using mkOrder.

- `defs`

  Function argument

lib.modules.mkIf
condition, content

Properties.

- `condition`

  Function argument

- `content`

  Function argument

lib.modules.fixMergeModules
modules, args

Compatibility.

- `modules`

  Function argument

- `args`

  Function argument

lib.modules.mkRemovedOptionModule
optionName, replacementInstructions, pattern, options, options

Return a module that causes a warning to be shown if the specified option is defined. For example,

- `optionName`

  Function argument

- `replacementInstructions`

  Function argument

- `pattern`

  Structured function argument

- `options`

  Function argument

- `options`

  Function argument

lib.modules.mkRenamedOptionModule
from, to

Return a module that causes a warning to be shown if the specified "from" option is defined; the defined value is however forwarded to the "to" option. This can be used to rename options while providing backward compatibility. For example,

- `from`

  Function argument

- `to`

  Function argument

lib.modules.mkMergedOptionModule
from, to, mergeFn, pattern, config, options, config, options

Return a module that causes a warning to be shown if any of the

"from" option is defined; the defined values can be used in the "mergeFn" to set the "to" value. This function can be used to merge multiple options into one that has a different type.

- `from`

  Function argument

- `to`

  Function argument

- `mergeFn`

  Function argument

- `pattern`

  Structured function argument

- `config`

  Function argument

- `options`

  Function argument

- `config`

  Function argument

- `options`

  Function argument

lib.modules.mkChangedOptionModule
from, to, changeFn

Single “from” version of mkMergedOptionModule. Return a module that causes a warning to be shown if the “from” option is defined; the defined value can be used in the “mergeFn” to set the “to” value. This function can be used to change an option into another that has a different type.

- `from`

  Function argument

- `to`

  Function argument

- `changeFn`

  Function argument

lib.modules.mkAliasOptionModule
from, to

Like ‘mkRenamedOptionModule’, but doesn’t show a warning.

- `from`

  Function argument

- `to`

  Function argument

lib.modules.mkDerivedConfig
opt, f

mkDerivedConfig : Option a -> (a -> Definition b) -> Definition b

- `opt`

  Function argument

- `f`

  Function argument

lib.modules.importJSON
file

Use this function to import a JSON file as NixOS configuration.

- `file`

  Function argument

lib.modules.importTOML
file

Use this function to import a TOML file as NixOS configuration.

- `file`

  Function argument

NixOS / nixpkgs option handling

lib.options.isOption

—

`isOption :: a -> bool`

Returns true when the given argument is an option

lib.options.isOption usage example:

```
isOption 1             // => false
isOption (mkOption {}) // => true
```

lib.options.mkOption
pattern, default, defaultText, example, description, relatedPackages, type, apply, internal, visible, readOnly, default, defaultText, example, description, relatedPackages, type, apply, internal, visible, readOnly

Creates an Option attribute set. mkOption accepts an attribute set with the following keys:

- `pattern`

  Structured function argument

- `default`

  Default value used when no definition is given in the configuration.

- `defaultText`

  Textual representation of the default, for the manual.

- `example`

  Example value used in the manual.

- `description`

  String describing the option.

- `relatedPackages`

Related packages used in the manual (see `genRelatedPackages` in ../nixos/lib/make-options-doc/default.nix).

- `type`

  Option type, providing type-checking and value merging.

- `apply`

  Function that converts the option value to something else.

- `internal`

  Whether the option is for NixOS developers only.

- `visible`

  Whether the option shows up in the manual. Default: true. Use false to hide the option and any sub-options from submodules. Use "shallow" to hide only sub-options.

- `readOnly`

  Whether the option can be set only once

- `default`

  Default value used when no definition is given in the configuration.

- `defaultText`

  Textual representation of the default, for the manual.

- `example`

Example value used in the manual.

- `description`

  String describing the option.

- `relatedPackages`

  Related packages used in the manual (see `genRelatedPackages` in ../nixos/lib/make-options-doc/default.nix).

- `type`

  Option type, providing type-checking and value merging.

- `apply`

  Function that converts the option value to something else.

- `internal`

  Whether the option is for NixOS developers only.

- `visible`

  Whether the option shows up in the manual. Default: true. Use false to hide the option and any sub-options from submodules. Use "shallow" to hide only sub-options.

- `readOnly`

  Whether the option can be set only once

lib.options.mkOption usage example:

```
mkOption { }  // => { _type = "option"; }
mkOption { default = "foo"; } // => { _type = "option";
default = "foo"; }
```

lib.options.mkEnableOption

name

Creates an Option attribute set for a boolean value option i.e an option to be toggled on or off:

- `name`

  Name for the created option

lib.options.mkEnableOption usage example:

```
mkEnableOption "foo"
=> { _type = "option"; default = false; description =
"Whether to enable foo."; example = true; type = { ... };
}
```

lib.options.mkPackageOption

pkgs, name, pattern, default, example, default, example

`mkPackageOption :: pkgs -> string -> { default :: [string], example :: null | string | [string] } -> optionThe package is specified as a list of strings representing its attribute path in nixpkgs.Because of this, you need to pass nixpkgs itself as the first argument.The second argument is the name of the option, used in the description "The <name> package to use.".You can also pass an example value, either a literal string or a package's attribute path.You can omit the default path if the name of the option is also attribute path in nixpkgs.`

Creates an Option attribute set for an option that specifies the package a module should use for some purpose.

- `pkgs`

  Package set (a specific version of nixpkgs)

- `name`

  Name for the package, shown in option description

- `pattern`

  Structured function argument

- `default`

  Function argument

- `example`

  Function argument

- `default`

  Function argument

- `example`

  Function argument

lib.options.mkPackageOption usage example:

```
mkPackageOption pkgs "hello" { }
=> { _type = "option"; default = «derivation /nix/store
/3r2vg51hlxj3cx5vscp0vkv60bqxkaq0-hello-2.10.drv»;
defaultText = { ... }; description = "The hello package
to use."; type = { ... }; }


mkPackageOption pkgs "GHC" {
default = [ "ghc" ];
example = "pkgs.haskell.packages.ghc924.ghc.withPackages
(hkgs: [ hkgs.primes ])";
}
=> { _type = "option"; default = «derivation /nix/store
/jxx55cxsjrf8kyh3fp2ya17q99w7541r-ghc-8.10.7.drv»;
defaultText = { ... }; description = "The GHC package to
use."; example = { ... }; type = { ... }; }
```

lib.options.mkSinkUndeclaredOptions
attrs

This option accepts anything, but it does not produce any result.

- `attrs`

  Function argument

lib.options.mergeEqualOption
loc, defs

“Merge” option definitions by checking that they all have the same value.

- `loc`

  Function argument

- `defs`

  Function argument

lib.options.getValues

—

`getValues :: [ { value :: a } ] -> [a]`

Extracts values of all “value” keys of the given list.

lib.options.getValues usage example:

```
getValues [ { value = 1; } { value = 2; } ] // => [ 1 2 ]
getValues [ ]                               // => [ ]
```

lib.options.getFiles

—

`getFiles :: [ { file :: a } ] -> [a]`

Extracts values of all “file” keys of the given list

lib.options.getFiles usage example:

```
getFiles [ { file = "file1"; } { file = "file2"; } ] //
=> [ "file1" "file2" ]
getFiles [ ]                                         //
=> [ ]
```

lib.options.scrubOptionValue

x

This function recursively removes all derivation attributes from `x` except for the `name` attribute.

- `x`

  Function argument

lib.options.literalExpression
text

For use in the `defaultText` and `example` option attributes. Causes the given string to be rendered verbatim in the documentation as Nix code. This is necessary for complex values, e.g. functions, or values that depend on other values or packages.

- `text`

  Function argument

lib.options.literalDocBook
text

For use in the `defaultText` and `example` option attributes. Causes the given DocBook text to be inserted verbatim in the documentation, for when a `literalExpression` would be too hard to read.

- `text`

  Function argument

lib.options.mdDoc
text

Transition marker for documentation that's already migrated to markdown syntax.

- `text`

Function argument

lib.options.literalMD

text

For use in the `defaultText` and `example` option attributes. Causes the given MD text to be inserted verbatim in the documentation, for when a `literalExpression` would be too hard to read.

- `text`

  Function argument

lib.options.showOption

parts

Convert an option, described as a list of the option parts in to a safe, human readable version.

- `parts`

  Function argument

lib.options.showOption usage example:

```
(showOption ["foo" "bar" "baz"]) == "foo.bar.baz"
(showOption ["foo" "bar.baz" "tux"]) == "foo.bar.baz.tux"

Placeholders will not be quoted as they are not actual
values:
(showOption ["foo" "*" "bar"]) == "foo.*.bar"
(showOption ["foo" "<name>" "bar"]) == "foo.<name>.bar"

Unlike attributes, options can also start with numbers:
(showOption ["windowManager" "2bwm" "enable"]) ==
"windowManager.2bwm.enable"
```

Source filtering functions

lib.sources.pathType

path

Returns the type of a path: regular (for file), symlink, or directory.

- `path`

  Function argument

lib.sources.pathIsDirectory

path

Returns true if the path exists and is a directory, false otherwise.

- `path`

  Function argument

lib.sources.pathIsRegularFile

path

Returns true if the path exists and is a regular file, false otherwise.

- `path`

  Function argument

lib.sources.cleanSourceFilter

name, type

A basic filter for `cleanSourceWith` that removes directories of version control system, backup files (*~) and some generated files.

- `name`

Function argument

- `type`

  Function argument

lib.sources.cleanSource
src

Filters a source tree removing version control files and directories using cleanSourceFilter.

- `src`

  Function argument

lib.sources.cleanSource usage example:

```
cleanSource ./.
```

lib.sources.cleanSourceWith
pattern, src, filter, name, src, filter, name

Like `builtins.filterSource`, except it will compose with itself, allowing you to chain multiple calls together without any intermediate copies being put in the nix store.

- `pattern`

  Structured function argument

- `src`

  A path or cleanSourceWith result to filter and/or rename.

- `filter`

Optional with default value: constant true (include everything)

- `name`

  Optional name to use as part of the store path.

- `src`

  A path or cleanSourceWith result to filter and/or rename.

- `filter`

  Optional with default value: constant true (include everything)

- `name`

  Optional name to use as part of the store path.

lib.sources.cleanSourceWith usage example:

```
lib.cleanSourceWith {
filter = f;
src = lib.cleanSourceWith {
filter = g;
src = ./.;
};
}
# Succeeds!

builtins.filterSource f (builtins.filterSource g ./.)
# Fails!
```

lib.sources.trace

src

`sources.trace :: sourceLike -> Source`

Add logging to a source, for troubleshooting the filtering behavior.

- `src`

  Source to debug. The returned source will behave like this source, but also log its filter invocations.

lib.sources.sourceByRegex
src, regexes

Filter sources by a list of regular expressions.

- `src`

  Function argument

- `regexes`

  Function argument

lib.sources.sourceByRegex usage example:

```
src = sourceByRegex ./my-subproject [".*\.py$"
"^database.sql$"]
```

lib.sources.sourceFilesBySuffices
src, exts

`sourceLike -> [String] -> Source`

Get all files ending with the specified suffices from the given source directory or its descendants, omitting files that do not match any suffix. The result of the example below will include files like `./dir/module.c` and `./dir/subdir/doc.xml` if present.

- `src`

Path or source containing the files to be returned

- `exts`

  A list of file suffix strings

lib.sources.sourceFilesBySuffices usage example:

```
sourceFilesBySuffices ./. [ ".xml" ".c" ]
```

lib.sources.commitIdFromGitRepo

—

Get the commit id of a git repo.

lib.sources.commitIdFromGitRepo usage example:

```
commitIdFromGitRepo <nixpkgs/.git>
```

String manipulation functions

lib.strings.concatStrings

—

`concatStrings :: [string] -> string`

Concatenate a list of strings.

lib.strings.concatStrings usage example:

```
concatStrings ["foo" "bar"]
=> "foobar"
```

lib.strings.concatMapStrings
f, list

`concatMapStrings :: (a -> string) -> [a] -> string`

Map a function over a list and concatenate the resulting strings.

- `f`

  Function argument

- `list`

  Function argument

lib.strings.concatMapStrings usage example:

```
concatMapStrings (x: "a" + x) ["foo" "bar"]
=> "afooabar"
```

lib.strings.concatImapStrings
f, list

`concatImapStrings :: (int -> a -> string) -> [a] -> string`

Like `concatMapStrings` except that the f functions also gets the position as a parameter.

- `f`

  Function argument

- `list`

  Function argument

lib.strings.concatImapStrings usage example:

```
concatImapStrings (pos: x: "${toString pos}-${x}") ["foo"
"bar"]
=> "1-foo2-bar"
```

lib.strings.intersperse

separator, *list*

`intersperse :: a -> [a] -> [a]`

Place an element between each element of a list

- `separator`

  Separator to add between elements

- `list`

  Input list

lib.strings.intersperse usage example:

```
intersperse "/" ["usr" "local" "bin"]
=> ["usr" "/" "local" "/" "bin"].
```

lib.strings.concatStringsSep

—

`concatStringsSep :: string -> [string] -> string`

Concatenate a list of strings with a separator between each element

lib.strings.concatStringsSep usage example:

```
concatStringsSep "/" ["usr" "local" "bin"]
=> "usr/local/bin"
```

lib.strings.concatMapStringsSep

sep, f, list

```
concatMapStringsSep :: string -> (a -> string) -> [a]
-> string
```

Maps a function over a list of strings and then concatenates the result with the specified separator interspersed between elements.

- `sep`

  Separator to add between elements

- `f`

  Function to map over the list

- `list`

  List of input strings

lib.strings.concatMapStringsSep usage example:

```
concatMapStringsSep "-" (x: toUpper x)  ["foo" "bar"
"baz"]
=> "FOO-BAR-BAZ"
```

lib.strings.concatImapStringsSep

sep, f, list

```
concatIMapStringsSep :: string -> (int -> a -> string)
-> [a] -> string
```

Same as `concatMapStringsSep`, but the mapping function additionally receives the position of its argument.

- `sep`

  Separator to add between elements

- `f`

  Function that receives elements and their positions

- `list`

  List of input strings

lib.strings.concatImapStringsSep usage example:

```
concatImapStringsSep "-" (pos: x: toString (x / pos)) [ 6
6 6 ]
=> "6-3-2"
```

lib.strings.makeSearchPath
subDir, paths

`makeSearchPath :: string -> [string] -> string`

Construct a Unix-style, colon-separated search path consisting of the given `subDir` appended to each of the given paths.

- `subDir`

  Directory name to append

- `paths`

  List of base paths

lib.strings.makeSearchPath usage example:

```
makeSearchPath "bin" ["/root" "/usr" "/usr/local"]
=> "/root/bin:/usr/bin:/usr/local/bin"
makeSearchPath "bin" [""]
=> "/bin"
```

lib.strings.makeSearchPathOutput

output, *subDir*, *pkgs*

`string -> string -> [package] -> string`

Construct a Unix-style search path by appending the given `subDir` to the specified `output` of each of the packages. If no output by the given name is found, fallback to `.out` and then to the default.

- `output`

  Package output to use

- `subDir`

  Directory name to append

- `pkgs`

  List of packages

lib.strings.makeSearchPathOutput usage example:

```
makeSearchPathOutput "dev" "bin" [ pkgs.openssl pkgs.zlib
]
=> "/nix/store/9rz8gxhzf8sw4kf2j2f1grr49w8zx5vj-openssl-
1.0.1r-dev/bin:/nix/store
/wwh7mhwh269sfjkm6k5665b5kgp7jrk2-zlib-1.2.8/bin"
```

lib.strings.makeLibraryPath

—

Construct a library search path (such as RPATH) containing the libraries for a set of packages

lib.strings.makeLibraryPath usage example:

```
makeLibraryPath [ "/usr" "/usr/local" ]
=> "/usr/lib:/usr/local/lib"
pkgs = import <nixpkgs> { }
makeLibraryPath [ pkgs.openssl pkgs.zlib ]
=> "/nix/store/9rz8gxhzf8sw4kf2j2f1grr49w8zx5vj-openssl-
1.0.1r/lib:/nix/store/wwh7mhwh269sfjkm6k5665b5kgp7jrk2-
zlib-1.2.8/lib"
```

lib.strings.makeBinPath

—

Construct a binary search path (such as $PATH) containing the binaries for a set of packages.

lib.strings.makeBinPath usage example:

```
makeBinPath ["/root" "/usr" "/usr/local"]
=> "/root/bin:/usr/bin:/usr/local/bin"
```

lib.strings.optionalString
cond, string

`optionalString :: bool -> string -> string`

Depending on the boolean `cond', return either the given string or the empty string. Useful to concatenate against a bigger string.

- `cond`

  Condition

- `string`

  String to return if condition is true

lib.strings.optionalString usage example:

```
optionalString true "some-string"
=> "some-string"
optionalString false "some-string"
=> ""
```

lib.strings.hasPrefix
pref, str

`hasPrefix :: string -> string -> bool`

Determine whether a string has given prefix.

- `pref`

  Prefix to check for

- `str`

  Input string

lib.strings.hasPrefix usage example:

```
hasPrefix "foo" "foobar"
=> true
hasPrefix "foo" "barfoo"
=> false
```

lib.strings.hasSuffix
suffix, content

`hasSuffix :: string -> string -> bool`

Determine whether a string has given suffix.

- `suffix`

  Suffix to check for

- `content`

  Input string

lib.strings.hasSuffix usage example:

```
hasSuffix "foo" "foobar"
=> false
hasSuffix "foo" "barfoo"
=> true
```

lib.strings.hasInfix
infix, content

`hasInfix :: string -> string -> bool`

Determine whether a string contains the given infix

- `infix`

  Function argument

- `content`

  Function argument

lib.strings.hasInfix usage example:

```
hasInfix "bc" "abcd"
=> true
hasInfix "ab" "abcd"
=> true
hasInfix "cd" "abcd"
=> true
hasInfix "foo" "abcd"
=> false
```

lib.strings.stringToCharacters

s

`stringToCharacters :: string -> [string]`

Convert a string to a list of characters (i.e. singleton strings). This allows you to, e.g., map a function over each character. However, note that this will likely be horribly inefficient; Nix is not a general purpose programming language. Complex string manipulations should, if appropriate, be done in a derivation. Also note that Nix treats strings as a list of bytes and thus doesn't handle unicode.

- `s`

  Function argument

lib.strings.stringToCharacters usage example:

```
stringToCharacters ""
=> [ ]
stringToCharacters "abc"
=> [ "a" "b" "c" ]
stringToCharacters "💩"
=> [ "" "" "" "" ]
```

lib.strings.stringAsChars

f, s

`stringAsChars :: (string -> string) -> string -> string`

Manipulate a string character by character and replace them by strings before concatenating the results.

- `f`

  Function to map over each individual character

- `s`

  Input string

lib.strings.stringAsChars usage example:

```
stringAsChars (x: if x == "a" then "i" else x) "nax"
=> "nix"
```

lib.strings.escape
list

`escape :: [string] -> string -> string`

Escape occurrence of the elements of `list` in `string` by prefixing it with a backslash.

- `list`

  Function argument

lib.strings.escape usage example:

```
escape ["(" ")"] "(foo)"
=> "\\(foo\\)"
```

lib.strings.escapeShellArg

arg

`escapeShellArg :: string -> string`

Quote string to be used safely within the Bourne shell.

- `arg`

  Function argument

lib.strings.escapeShellArg usage example:

```
escapeShellArg "esc'ape\nme"
=> "'esc'\\''ape\nme'"
```

lib.strings.escapeShellArgs

—

`escapeShellArgs :: [string] -> string`

Quote all arguments to be safely passed to the Bourne shell.

lib.strings.escapeShellArgs usage example:

```
escapeShellArgs ["one" "two three" "four'five"]
=> "'one' 'two three' 'four'\\''five'"
```

lib.strings.isValidPosixName

name

`string -> bool`

Test whether the given name is a valid POSIX shell variable name.

- `name`

Function argument

lib.strings.isValidPosixName usage example:

```
isValidPosixName "foo_bar000"
=> true
isValidPosixName "0-bad.jpg"
=> false
```

lib.strings.toShellVar

name, value

`string -> (string | listOf string | attrsOf string) -> string`

Translate a Nix value into a shell variable declaration, with proper escaping.

- `name`

  Function argument

- `value`

  Function argument

lib.strings.toShellVar usage example:

```
''
${toShellVar "foo" "some string"}
[[ "$foo" == "some string" ]]
''
```

lib.strings.toShellVars

vars

`attrsOf (string | listOf string | attrsOf string) -> string`

Translate an attribute set into corresponding shell variable declarations using `toShellVar`.

- `vars`

  Function argument

lib.strings.toShellVars usage example:

```
let
foo = "value";
bar = foo;
in ''
${toShellVars { inherit foo bar; }}
[[ "$foo" == "$bar" ]]
''
```

lib.strings.escapeNixString

s

`string -> string`

Turn a string into a Nix expression representing that string

- `s`

  Function argument

lib.strings.escapeNixString usage example:

```
escapeNixString "hello\${}\n"
=> "\"hello\\\${}\\n\""
```

lib.strings.escapeRegex

—

`string -> string`

Turn a string into an exact regular expression

lib.strings.escapeRegex usage example:

```
escapeRegex "[^a-z]*"
=> "\\[\\^a-z]\\*"
```

lib.strings.escapeNixIdentifier

s

`string -> string`

Quotes a string if it can't be used as an identifier directly.

- `s`

  Function argument

lib.strings.escapeNixIdentifier usage example:

```
escapeNixIdentifier "hello"
=> "hello"
escapeNixIdentifier "0abc"
=> "\"0abc\""
```

lib.strings.escapeXML

—

`string -> string`

Escapes a string such that it is safe to include verbatim in an XML

document.

lib.strings.escapeXML usage example:

```
escapeXML ''"test" 'test' < & >''
=> "&quot;test&quot; &apos;test&apos; &lt; &amp; &gt;"
```

lib.strings.toLower

—

`toLower :: string -> string`

Converts an ASCII string to lower-case.

lib.strings.toLower usage example:

```
toLower "HOME"
=> "home"
```

lib.strings.toUpper

—

`toUpper :: string -> string`

Converts an ASCII string to upper-case.

lib.strings.toUpper usage example:

```
toUpper "home"
=> "HOME"
```

lib.strings.addContextFrom
a, b

Appends string context from another string. This is an implementation detail of Nix.

- `a`

  Function argument

- `b`

  Function argument

lib.strings.addContextFrom usage example:

```
pkgs = import <nixpkgs> { };
addContextFrom pkgs.coreutils "bar"
=> "bar"
```

lib.strings.splitString

__sep, *s*

Cut a string with a separator and produces a list of strings which were separated by this separator.

- `_sep`

  Function argument

- `_s`

  Function argument

lib.strings.splitString usage example:

```
splitString "." "foo.bar.baz"
=> [ "foo" "bar" "baz" ]
splitString "/" "/usr/local/bin"
=> [ "" "usr" "local" "bin" ]
```

lib.strings.removePrefix

prefix, str

`string -> string -> string`

Return a string without the specified prefix, if the prefix matches.

- `prefix`

  Prefix to remove if it matches

- `str`

  Input string

lib.strings.removePrefix usage example:

```
removePrefix "foo." "foo.bar.baz"
=> "bar.baz"
removePrefix "xxx" "foo.bar.baz"
=> "foo.bar.baz"
```

lib.strings.removeSuffix
suffix, str

`string -> string -> string`

Return a string without the specified suffix, if the suffix matches.

- `suffix`

  Suffix to remove if it matches

- `str`

  Input string

lib.strings.removeSuffix usage example:

```
removeSuffix "front" "homefront"
=> "home"
removeSuffix "xxx" "homefront"
=> "homefront"
```

lib.strings.versionOlder
v1, v2

Return true if string v1 denotes a version older than v2.

- `v1`

  Function argument

- `v2`

  Function argument

lib.strings.versionOlder usage example:

```
versionOlder "1.1" "1.2"
=> true
versionOlder "1.1" "1.1"
=> false
```

lib.strings.versionAtLeast
v1, v2

Return true if string v1 denotes a version equal to or newer than v2.

- `v1`

  Function argument

- v2

  Function argument

lib.strings.versionAtLeast usage example:

```
versionAtLeast "1.1" "1.0"
=> true
versionAtLeast "1.1" "1.1"
=> true
versionAtLeast "1.1" "1.2"
=> false
```

lib.strings.getName

x

This function takes an argument that's either a derivation or a derivation's "name" attribute and extracts the name part from that argument.

- x

  Function argument

lib.strings.getName usage example:

```
getName "youtube-dl-2016.01.01"
=> "youtube-dl"
getName pkgs.youtube-dl
=> "youtube-dl"
```

lib.strings.getVersion

x

This function takes an argument that's either a derivation or a derivation's "name" attribute and extracts the version part from

that argument.

- `x`

  Function argument

lib.strings.getVersion usage example:

```
getVersion "youtube-dl-2016.01.01"
=> "2016.01.01"
getVersion pkgs.youtube-dl
=> "2016.01.01"
```

lib.strings.nameFromURL
url, sep

Extract name with version from URL. Ask for separator which is supposed to start extension.

- `url`

  Function argument

- `sep`

  Function argument

lib.strings.nameFromURL usage example:

```
nameFromURL "https://nixos.org/releases/nix/nix-1.7/nix-
1.7-x86_64-linux.tar.bz2" "-"
=> "nix"
nameFromURL "https://nixos.org/releases/nix/nix-1.7/nix-
1.7-x86_64-linux.tar.bz2" "_"
=> "nix-1.7-x86"
```

lib.strings.enableFeature
enable, feat

Create an –{enable,disable}- string that can be passed to standard GNU Autoconf scripts.

- `enable`

  Function argument

- `feat`

  Function argument

lib.strings.enableFeature usage example:

```
enableFeature true "shared"
=> "--enable-shared"
enableFeature false "shared"
=> "--disable-shared"
```

lib.strings.enableFeatureAs
enable, feat, value

Create an –{enable-=,disable-} string that can be passed to standard GNU Autoconf scripts.

- `enable`

  Function argument

- `feat`

  Function argument

- `value`

Function argument

lib.strings.enableFeatureAs usage example:

```
enableFeatureAs true "shared" "foo"
=> "--enable-shared=foo"
enableFeatureAs false "shared" (throw "ignored")
=> "--disable-shared"
```

lib.strings.withFeature
with, feat_

Create an –{with,without}- string that can be passed to standard GNU Autoconf scripts.

- `with_`

  Function argument

- `feat`

  Function argument

lib.strings.withFeature usage example:

```
withFeature true "shared"
=> "--with-shared"
withFeature false "shared"
=> "--without-shared"
```

lib.strings.withFeatureAs
with, feat, value_

Create an –{with-=,without-} string that can be passed to standard GNU Autoconf scripts.

- `with_`

  Function argument

- `feat`

  Function argument

- `value`

  Function argument

lib.strings.withFeatureAs usage example:

```
withFeatureAs true "shared" "foo"
=> "--with-shared=foo"
withFeatureAs false "shared" (throw "ignored")
=> "--without-shared"
```

lib.strings.fixedWidthString
width, filler, str

`fixedWidthString :: int -> string -> string -> string`

Create a fixed width string with additional prefix to match required width.

- `width`

  Function argument

- `filler`

  Function argument

- `str`

Function argument

lib.strings.fixedWidthString usage example:

```
fixedWidthString 5 "0" (toString 15)
=> "00015"
```

lib.strings.fixedWidthNumber
width, n

Format a number adding leading zeroes up to fixed width.

- `width`

  Function argument

- `n`

  Function argument

lib.strings.fixedWidthNumber usage example:

```
fixedWidthNumber 5 15
=> "00015"
```

lib.strings.floatToString
float

Convert a float to a string, but emit a warning when precision is lost during the conversion

- `float`

  Function argument

lib.strings.floatToString usage example:

```
floatToString 0.000001
=> "0.000001"
floatToString 0.0000001
=> trace: warning: Imprecise conversion from float to
string 0.000000
"0.000000"
```

lib.strings.isCoercibleToString

x

Check whether a value can be coerced to a string

- x

  Function argument

lib.strings.isStorePath

x

Check whether a value is a store path.

- x

  Function argument

lib.strings.isStorePath usage example:

```
isStorePath "/nix/store/d945ibfx9x185xf04b890y4f9g3cbb63-
python-2.7.11/bin/python"
=> false
isStorePath "/nix/store/d945ibfx9x185xf04b890y4f9g3cbb63-
python-2.7.11"
=> true
isStorePath pkgs.python
=> true
isStorePath [] || isStorePath 42 || isStorePath {} || …
=> false
```

lib.strings.toInt

str

`string -> int`

Parse a string as an int.

- `str`

  Function argument

lib.strings.toInt usage example:

```
toInt "1337"
=> 1337
toInt "-4"
=> -4
toInt "3.14"
=> error: floating point JSON numbers are not supported
```

lib.strings.readPathsFromFile

—

Read a list of paths from `file`, relative to the `rootPath`. Lines beginning with `#` are treated as comments and ignored.

Whitespace is significant.

lib.strings.readPathsFromFile usage example:

```
readPathsFromFile /prefix
./pkgs/development/libraries/qt-5/5.4/qtbase/series
=> [ "/prefix/dlopen-resolv.patch" "/prefix/tzdir.patch"
"/prefix/dlopen-libXcursor.patch" "/prefix/dlopen-
openssl.patch"
"/prefix/dlopen-dbus.patch" "/prefix/xdg-config-
dirs.patch"
"/prefix/nix-profiles-library-paths.patch"
"/prefix/compose-search-path.patch" ]
```

lib.strings.fileContents

file

`fileContents :: path -> string`

Read the contents of a file removing the trailing

- `file`

  Function argument

lib.strings.fileContents usage example:

```
$ echo "1.0" > ./version

fileContents ./version
=> "1.0"
```

lib.strings.sanitizeDerivationName

—

`sanitizeDerivationName :: String -> String`

Creates a valid derivation name from a potentially invalid one.

lib.strings.sanitizeDerivationName usage example:

```
sanitizeDerivationName "../hello.bar # foo"
=> "-hello.bar-foo"
sanitizeDerivationName ""
=> "unknown"
sanitizeDerivationName pkgs.hello
=> "-nix-store-2g75chlbpxlrqn15zlby2dfh8hr9qwbk-
hello-2.10"
```

lib.strings.levenshtein
a, b

`levenshtein :: string -> string -> int`

Computes the Levenshtein distance between two strings.
Complexity O(n*m) where n and m are the lengths of the strings.
Algorithm adjusted from https://stackoverflow.com/a/9750974
/6605742

- `a`

  Function argument

- `b`

  Function argument

lib.strings.levenshtein usage example:

```
levenshtein "foo" "foo"
=> 0
levenshtein "book" "hook"
=> 1
levenshtein "hello" "Heyo"
=> 3
```

lib.strings.commonPrefixLength

a, *b*

Returns the length of the prefix common to both strings.

- `a`

  Function argument

- `b`

  Function argument

lib.strings.commonSuffixLength

a, *b*

Returns the length of the suffix common to both strings.

- `a`

  Function argument

- `b`

  Function argument

lib.strings.levenshteinAtMost

—

```
levenshteinAtMost :: int -> string -> string -> bool
```

Returns whether the levenshtein distance between two strings is at most some value Complexity is O(min(n,m)) for k <= 2 and O(n*m) otherwise

lib.strings.levenshteinAtMost usage example:

```
levenshteinAtMost 0 "foo" "foo"
=> true
levenshteinAtMost 1 "foo" "boa"
=> false
levenshteinAtMost 2 "foo" "boa"
=> true
levenshteinAtMost 2 "This is a sentence" "this is a
sentense."
=> false
levenshteinAtMost 3 "This is a sentence" "this is a
sentense."
=> true
```

Miscellaneous functions

lib.trivial.id

x

```
id :: a -> a
```

The identity function For when you need a function that does “nothing”.

- `x`

  The value to return

lib.trivial.const

x, y

`const :: a -> b -> a`

The constant function

- `x`

  Value to return

- `y`

  Value to ignore

lib.trivial.const usage example:

```
let f = const 5; in f 10
=> 5
```

lib.trivial.pipe
val, functions

`pipe :: a -> [<functions>] -> <return type of last function>`

Pipes a value through a list of functions, left to right.

- `val`

  Function argument

- `functions`

  Function argument

lib.trivial.pipe usage example:

```
pipe 2 [
(x: x + 2)  # 2 + 2 = 4
(x: x * 2)  # 4 * 2 = 8
]
=> 8

# ideal to do text transformations
pipe [ "a/b" "a/c" ] [

# create the cp command
(map (file: ''cp "${src}/${file}" $out\n''))

# concatenate all commands into one string
lib.concatStrings

# make that string into a nix derivation
(pkgs.runCommand "copy-to-out" {})

]
=> <drv which copies all files to $out>

The output type of each function has to be the input type
of the next function, and the last function returns the
final value.
```

lib.trivial.concat

x, y

`concat :: [a] -> [a] -> [a]`

Concatenate two lists

- x

  Function argument

- y

  Function argument

lib.trivial.concat usage example:

```
concat [ 1 2 ] [ 3 4 ]
=> [ 1 2 3 4 ]
```

lib.trivial.or

x, *y*

boolean “or”

- x

  Function argument

- y

  Function argument

lib.trivial.and

x, *y*

boolean “and”

- x

  Function argument

- y

  Function argument

lib.trivial.bitAnd

—

bitwise “and”

lib.trivial.bitOr

—

bitwise “or”

lib.trivial.bitXor

—

bitwise “xor”

lib.trivial.bitNot

—

bitwise “not”

lib.trivial.boolToString
b

```
boolToString :: bool -> string
```

Convert a boolean to a string.

- b

  Function argument

lib.trivial.mergeAttrs
x, *y*

Merge two attribute sets shallowly, right side trumps left

- x

Left attribute set

- `y`

  Right attribute set (higher precedence for equal keys)

lib.trivial.mergeAttrs usage example:

```
mergeAttrs { a = 1; b = 2; } { b = 3; c = 4; }
=> { a = 1; b = 3; c = 4; }
```

lib.trivial.flip

f, *a*, *b*

```
flip :: (a -> b -> c) -> (b -> a -> c)
```

Flip the order of the arguments of a binary function.

- `f`

  Function argument

- `a`

  Function argument

- `b`

  Function argument

lib.trivial.flip usage example:

```
flip concat [1] [2]
=> [ 2 1 ]
```

lib.trivial.mapNullable

f, *a*

Apply function if the supplied argument is non-null.

- `f`

  Function to call

- `a`

  Argument to check for null before passing it to `f`

lib.trivial.mapNullable usage example:

```
mapNullable (x: x+1) null
=> null
mapNullable (x: x+1) 22
=> 23
```

lib.trivial.version

—

Returns the current full nixpkgs version number.

lib.trivial.release

—

Returns the current nixpkgs release number as string.

lib.trivial.oldestSupportedRelease

—

The latest release that is supported, at the time of release branch-off, if applicable.

lib.trivial.isInOldestRelease

release

Whether a feature is supported in all supported releases (at the time of release branch-off, if applicable). See `oldestSupportedRelease`.

- `release`

  Release number of feature introduction as an integer, e.g. 2111 for 21.11. Set it to the upcoming release, matching the nixpkgs/.version file.

lib.trivial.codeName

—

Returns the current nixpkgs release code name.

lib.trivial.versionSuffix

—

Returns the current nixpkgs version suffix as string.

lib.trivial.revisionWithDefault
default

```
revisionWithDefault :: string -> string
```

Attempts to return the the current revision of nixpkgs and returns the supplied default value otherwise.

- `default`

  Default value to return if revision can not be determined

lib.trivial.inNixShell

—

`inNixShell :: bool`

Determine whether the function is being called from inside a Nix shell.

lib.trivial.inPureEvalMode

—

`inPureEvalMode :: bool`

Determine whether the function is being called from inside pure-eval mode by seeing whether `builtins` contains `currentSystem`. If not, we must be in pure-eval mode.

lib.trivial.min
x, y

Return minimum of two numbers.

- x

  Function argument

- y

  Function argument

lib.trivial.max
x, y

Return maximum of two numbers.

- x

  Function argument

- `y`

  Function argument

lib.trivial.mod
base, int

Integer modulus

- `base`

  Function argument

- `int`

  Function argument

lib.trivial.mod usage example:

```
mod 11 10
=> 1
mod 1 10
=> 1
```

lib.trivial.compare
a, b

C-style comparisons

- `a`

  Function argument

- `b`

  Function argument

lib.trivial.splitByAndCompare

p, yes, no, a, b

```
(a -> bool) -> (a -> a -> int) -> (a -> a -> int) -> (a -> a -> int)
```

Split type into two subtypes by predicate `p`, take all elements of the first subtype to be less than all the elements of the second subtype, compare elements of a single subtype with `yes` and `no` respectively.

- `p`

  Predicate

- `yes`

  Comparison function if predicate holds for both values

- `no`

  Comparison function if predicate holds for neither value

- `a`

  First value to compare

- `b`

  Second value to compare

lib.trivial.splitByAndCompare usage example:

```
let cmp = splitByAndCompare (hasPrefix "foo") compare
compare; in

cmp "a" "z" => -1
cmp "fooa" "fooz" => -1

cmp "f" "a" => 1
cmp "fooa" "a" => -1
# while
compare "fooa" "a" => 1
```

lib.trivial.importJSON
path

Reads a JSON file.

- `path`

  Function argument

lib.trivial.importTOML
path

Reads a TOML file.

- `path`

  Function argument

lib.trivial.warn

—

`string -> a -> a`

Print a warning before returning the second argument. This function behaves like `builtins.trace`, but requires a string

message and formats it as a warning, including the `warning:` prefix.

lib.trivial.warnIf
cond, msg

`bool -> string -> a -> a`

Like warn, but only warn when the first argument is `true`.

- `cond`

  Function argument

- `msg`

  Function argument

lib.trivial.warnIfNot
cond, msg

`bool -> string -> a -> a`

Like warnIf, but negated (warn if the first argument is `false`).

- `cond`

  Function argument

- `msg`

  Function argument

lib.trivial.throwIfNot
cond, msg

`bool -> string -> a -> a`

Like the `assert b; e` expression, but with a custom error message and without the semicolon.

- `cond`

  Function argument

- `msg`

  Function argument

lib.trivial.throwIfNot usage example:

```
throwIfNot (lib.isList overlays) "The overlays argument
to nixpkgs must be a list."
lib.foldr (x: throwIfNot (lib.isFunction x) "All overlays
passed to nixpkgs must be functions.") (r: r) overlays
pkgs
```

lib.trivial.throwIf
cond, msg

`bool -> string -> a -> a`

Like throwIfNot, but negated (throw if the first argument is `true`).

- `cond`

  Function argument

- `msg`

  Function argument

lib.trivial.checkListOfEnum

msg, valid, given

`String -> List ComparableVal -> List ComparableVal -> a -> a`

Check if the elements in a list are valid values from a enum, returning the identity function, or throwing an error message otherwise.

- `msg`

  Function argument

- `valid`

  Function argument

- `given`

  Function argument

lib.trivial.checkListOfEnum usage example:

```
let colorVariants = ["bright" "dark" "black"]
in checkListOfEnum "color variants" [ "standard" "light"
"dark" ] colorVariants;
=>
error: color variants: bright, black unexpected; valid
ones: standard, light, dark
```

lib.trivial.setFunctionArgs

f, args

Add metadata about expected function arguments to a function. The metadata should match the format given by

builtins.functionArgs, i.e. a set from expected argument to a bool representing whether that argument has a default or not.
setFunctionArgs : (a → b) → Map String Bool → (a → b)

- `f`

  Function argument

- `args`

  Function argument

lib.trivial.functionArgs
f

Extract the expected function arguments from a function. This works both with nix-native { a, b ? foo, … }: style functions and functions with args set with 'setFunctionArgs'. It has the same return type and semantics as builtins.functionArgs.
setFunctionArgs : (a → b) → Map String Bool.

- `f`

  Function argument

lib.trivial.isFunction
f

Check whether something is a function or something annotated with function args.

- `f`

  Function argument

lib.trivial.toFunction

v

Turns any non-callable values into constant functions. Returns callable values as is.

- `v`

  Any value

lib.trivial.toFunction usage example:

```
nix-repl> lib.toFunction 1 2
1

nix-repl> lib.toFunction (x: x + 1) 2
3
```

lib.trivial.toHexString
i

Convert the given positive integer to a string of its hexadecimal representation. For example:

- `i`

  Function argument

lib.trivial.toBaseDigits
base, i

`toBaseDigits base i` converts the positive integer i to a list of its digits in the given base. For example:

- `base`

  Function argument

- `i`

  Function argument

Versions functions

lib.versions.splitVersion

—

Break a version string into its component parts.

lib.versions.splitVersion usage example:

```
splitVersion "1.2.3"
=> ["1" "2" "3"]
```

lib.versions.major

v

Get the major version string from a string.

- `v`

  Function argument

lib.versions.major usage example:

```
major "1.2.3"
=> "1"
```

lib.versions.minor

v

Get the minor version string from a string.

- `v`

Function argument

lib.versions.minor usage example:

```
minor "1.2.3"
=> "2"
```

lib.versions.patch

v

Get the patch version string from a string.

- `v`

  Function argument

lib.versions.patch usage example:

```
patch "1.2.3"
=> "3"
```

lib.versions.majorMinor

v

Get string of the first two parts (major and minor) of a version string.

- `v`

  Function argument

lib.versions.majorMinor usage example:

```
majorMinor "1.2.3"
=> "1.2"
```

Theme: jez/pandoc-markdown-css-theme