

Design documents

This category contains design documents written by the Lix team, which may or may not be implemented.

- [regex engine investigation](#)
- [Dreams](#)
- [Language versioning](#)
- [Docs rewrite plans](#)

regex engine investigation

nix uses libstdc++'s `std::regex`. it uses whatever version of libstdc++ the host system has.

which it invokes in both `std::regex_replace` `std::regex_match` modes.

nix occasionally uses the flags `std::regex::extended` and `std::regex::icase` which determine the available features - it's always either no flags, or both of these together. there's also a couple things that use the flag `std::regex::ECMAScript`. when the constructor is called without a flags parameter, the flags default to `std::regex::ECMAScript` (see method signature in C++23 32.7.2), so really we have only two cases.

`std::cregex_iterator` and `std::sregex_iterator` are used.

there's a header `regex-combinators.hh` which defines `regex::group` and `regex::list`.... and a couple others that are unused. but those are just trivial textual things, not extensions, so we can ignore the file.

getting the C++ standard

someday when C++23 is official you will be able to pirate the PDF. otherwise, you can clone <https://github.com/cplusplus/draft> and check out the tag `n4950` which is the current formally adopted working draft as of 2024-03-14 and is intended to have the same technical content as the final standard. you can then invoke `make` in the `source` subdirectory which will produce `std.pdf`. you will need LaTeX installed. if you're ever not sure which working draft is the one that became a particular version of the standard, Wikipedia will probably tell you...

(personally I install `texlive.combined.scheme-full` from nixpkgs on all my machines that have room for it, but this is surely more than necessary, it just makes me feel warm and fuzzy -- Irenes)

chapter 32 is the one that documents regular expressions.

open questions that require reading the standard

- what are all the syntactic and semantic constructs we need to support?

required functionality

the `extended` flag, per the C++ standard, "Specifies that the grammar recognized by the regular expression engine shall be that used by extended regular expressions in POSIX.". it references POSIX, Base Definitions and Headers, Section 9.4.

the `ECMAScript` flag "Specifies that the grammar recognized by the regular expression engine shall be that used by ECMAScript in ECMA-262, as modified in [section 32.12 of the C++ standard]." it references ECMA-262 15.10. the changes in 32.12 are important and probably do create real compatibility issues for us, though fortunately it's only a single page.

if we complete this chart we can use it to assess which existing engines would meet our needs, or how much of a pain in the ass it would be to make a new one

the columns are the two ways it gets invoked

	extended + icode	ECMAScript
Syntactic constructs	--	--
(TODO: fill in every construct here)		
Semantics	--	--
Case-insensitivity	yes	?
(TODO: fill in other behaviors here)		

Dreams

This page documents the dreams of the Lix team. These are features which we have generally not roadmapped yet, and which may not have complete and thoroughly thought-through plans, and which we would like to think about more completely before implementing. We are writing them down publicly so that others can dream with us.

- language versioning <https://pad.lix.systems/lix-lang-version-investigation> [PRIVATE LINK]
- split the evaluator into a separate process, interact with it only via rpc (horrors)
- bytecode evaluator with all the possible trappings (horrors)
 - allows arbitrary runtime-define breakpoints, watchpoints, program inspection and manipulation
 - interacts with rpc to allow perfect lsp hosts, better debuggers etc
- new gc for the evaluator to replace bdw, prototype/template for gc in eventual rust evaluator (horrors)
- flakes as a library of code that provides new nix subcommands (horrors, others)
- lix.conf `prelude-path =` for system-wide subcommands a la git (horrors)
 - also can make per-repo `lix *` commands (jade, janik)
- eval caching with a `memoize :: str -> any -> any` builtin that is overridden by `scopedImport` with a unique, deterministic subscope (horrors)
 - `import := f: memoize (toString f) (scopedImport builtins f)` (horrors)
- flake eval caching entire attrpaths: `mapAttrsRecursive (n: const (memoize n))` on all scopes/attrsets in the "flake" (horrors)
- lazyUpdate is a disaster waiting to happen, turns all values into even worse errors sources than simple thunks (and is deeply intrusive to the evaluator for little gain). why not special attrset ops `__members`, `__getMember` to simulate lazyUpdate in a library that doesn't infect all future versions of the language and can be transpiled when necessary? (horrors)
 - pureImport is too fine grained, store paths as boundaries actually make sense (and give memoize stable starting scopes), pure eval mode could be "ask thing to pack itself up, add to store, eval from there like nix flakes do" (horrors)
- all authoritative information about the store attached to store objects, not an sqlite database (eg in xattrs or similar) (horrors)
 - would make overlayfs stores for containers/vms trivial
- redo the lazy trees infra on the basis of "virtual" store paths and mountpoints (turning eg a zip file into a virtual mountpoint `/nix/store/lazy/thing.zip/...`) (horrors)
 - notably do not use fuse for this, just a pure vfs implementation
- fully decouple evaluator and store (horrors)
 - tvix has kind of done this with EvalIO, lix needs it too (otherwise the eval-process split will not be possible)
- store operations state, like "what derivations were realized in the last build" (Qyriad)
 - "what attrpath was this accessed by to build"
- profiler for nix code (jade)
- nix develop replace store path but actually good, with bind mounts (jade)

- nixos-rebuild gets unfucked perhaps with samueldr code (jade)
- we kinda wanna have inherits consistent by container type such that you can write inherit (thing) [a b c] to create a list, inherit (thing) { a b c } to create a set, or nest those in existing lists or sets to extend them in-place like current inherit (horrors)
- unbreak the io model (horrors)
 - currently nix has an async io model shoved into a sync runtime, and an async model that can't decide whether it's push or pull. this **sucks**
- a dependency graph for builds which explains why different dependencies are being built
 - store path truncated to unique names in output...?
- native nix-output-monitor (`nom`) style (slash bazel-style) output formatting (showing a live updating list of stuff being built/fetched, with warnings stacking up above it)
 - web viewer for the build graph as it is happening with a nice live log viewer (jade)
 - relatedly: show closure graphs nicely (jade)
- make the store properly multi-tenant, with things like, e.g. authentication and maybe even certain http done via hooks on the client side (jade)
 - see e.g. <https://git.lix.systems/lix-project/lix/issues/254>
 - overall improve the clarity of what is actually running on the daemon vs the client (jade)
- replace `nix profile` with something not broken with a clear ramp to either have a manifest mutably in the store or operate mutably against a configuration directory. ideally out of tree. (jade)
- fix fs builtin problems (jade)
 - can't read symlinks
 - filterSource gives no metadata of interest esp on symlinks
 - can't synthesize symlinks or files into the store except by serious nar abuse
 - (Zoe) We can imagine a generalized transformSource builtin which presents an fs subtree as a nested attrSet containing the full metadata and contents of all files and links in the subtree, and expects an nested attrSet in the same format as output, allowing arbitrary transformations in pure nix code. As long any other other operations that touch touch the file system are disallowed inside the transformation function (evaluating other paths, building derivations, pathExists, etc) this should be a consistent operation. There may be performance/usability reasons to not use this precise interface, but I think it's a good abstract guide stone of what to strive for.
 - is lib.filesets made of evil? how does it work?
 - answer: it's filterSource in a trench coat with some set operations
 - what if you could take a source tree of a monorepo and rewrite cross project symlinks to refer to store paths of those other projects so you don't copy the entire giant repo to store every time and can have each subproject as its own store path?
 - what if you had a fetch git subtree primitive that was free if there's no modification?
 - (Zoe) It's a little trickier than just that because if you want a filtered git subtree you need some way to ensure that the filter hasn't changed either.
- Better facilities for writing performant code (Zoe)
 - Builtins should document their algorithmics and when they cause files to be written to the store

- More opt-in persistent data structures with different performance tradeoffs that can be coerced to from the standard values
 - RRB vectors or similar for lists
 - HAMT or similar for attrSets
 - should allow using arbitrary values as keys
 - will probably need an explicit distinction between strings and symbols
 - also a separate set type, so you don't have to bother faking it with null keys
 - StringView like type for strings
 - or maybe just convert in place the first time we'd need to get the length?
- Doing something about IFD being bad (raito, pennae?):
<https://pad.lix.systems/sW0nbPohTgqy2UdIjPeUA>

fixing ux

- some way of having a persistent short lived evaluator for fast completions in CLI (Dawn)
- `□` fancy `□` repl, a la IPython and pry (Qyriad)
- Support instance of Lix running locally off the main page to try out
 - Obviously WebAssembly schenanigans involved
- replacing nixos-option (jade)
 - CLI commands should be possible to actually deprecate (jade)
- a debug macro like rust's `dbg!` <https://doc.rust-lang.org/std/macro.dbg.html>
- pipe operator (Qyriad)
 - and either haskell's `$` or left pipe operator
- hyperlinked sources in docs (jade)
- a VFS mirror of the Nix store that puts the names first, attaches a more descriptive label if necessary, and then the hash, literally just for convenience (Qyriad)

slaying the hydra

these are problems that make hydra sad

- make -jsem jobserver built into Lix (horrors actually wrote one years ago)
 - this would allow much better build density in Lix and eliminate most need to tune `NIX_BUILD_CORES`
 - see: <https://github.com/NixOS/nixpkgs/pull/143820>, it turns out the make jobserver protocol is actually *horrible*, and we should instead do this with a reasonable socket protocol injected into the sandbox by Lix
- externalize deciding which host to build things on (delroth, jade)
 - this is necessary because `/etc/nix/machines` is really stupid and doesn't have nearly enough information to decide whether a machine can admit a job.
- make the remote protocol not suck (jade)
 - latency is bad

- a lot of stuff blocks in ways it only dubiously needs to?
- what if you could have build cost estimates on large installations, which could go into scheduling decisions? (jade)
 - galaxy brained idea: build a neural net for derivation build costs for scheduling purposes. probably take as input the `derivation show` json with the hashes removed and then a pile of historical hydra data
 - do we have the data to do this? we want cpu time, io, and (ugh these would be very fake though because measuring memory is fraught) memory stats for builds.
 - schedule on machines that have space for the expected cpu-time/memory-time/io-time of the derivation
- make the nix daemon know what is actually building (jade)

Language versioning

This document is extremely a draft. It needs some editing and discussion before it can be made into a useful thing. It's been simply copy pasted out of the pad in its current form.

See also

- FIXME: piegames langver ideas

musings

puck: honestly, having language version as part of a `scopedImport`-style primop would be funny
horrors: we're shitposting about setting language version from the source accessor

- horrors: `use features...;` file head clause
 - jade: this can be combined into feature sets like editions or such. we might become ghc haskell but whatever.
- horrors: [some kind of file head clause and/or propagation is the] only real way out of this mess that doesn't require a package manager in the package manager
jade: yeah. doing it from flakes seems initially sane until you realise you can import below the flake in the same git repo and then blegh
horrors: an ambient minimum-language-version binding in builtins that can be `scopedImport`'ed for flake support on top of this

horrific writeup

basic mechanism

add a new syntactic element that is only valid at the head of a file and used only to declare language requirements. nix versions that cannot satisfy all requirements must reject this element to situations in which two nix versions parse the same file differently, or even evaluating the same file to different derivation hashes. any kind of comment as used by eg GHC is not viable for nix for this reason.

proposed syntax for the first implementation: `use $($feature: ident)+;`

anything ahead of this directive could be either unversioned nix code or versioned nix code (see below for details), but since the directive is only valid at the *head* of a file or expression this "code" can only be comments. this kind of locks us into supporting the current comment syntax forever,

but the comment syntax is rather fine so this won't be a problem.

each feature may declare a syntactical requirement for the file, a semantic requirement, or possible both (cf rust editions, or perl `use v<something>`).

features may be global, namespaced to their implementations, or live in a reserved `experimental` namespace an implementation can add to and remove from as it wishes with ***absolutely no guarantee of future evaluatility***.

syntactic features

syntax is entirely local to the file itself and has few to no intercompatibility constraints with other code. a very useful syntax requirement would something like `no-url-literals`, which might strip the syntactic ability to parse url-like sequences of characters into strings and, rather than nix currently does the experimental feature of the same name simply throwing a parse error, parse them as eg a lambda with a sequence of divisions in its body.

(realistically `no-url-literals` would not appear in practice, instead it should be implied by `use` itself since url literals are such an obvious misfeature)

semantic features

semantic features produce evaluation changes that could be achieved any other way. examples of this are:

- the recent change that evaluates `x` in `inherit (x) names...` at most once overall rather than once per inherited name accessed
- potential extensions to the string context mechanism
- new types of values

semantic changes may escape the expression that requires them and usually some of amount of cross-compatibility with other semantic versions must be given. using the same examples as above, considerations can include:

- observable side-effects changing (if `x` includes a call to `trace`)
- `getContext` returning sets an outside use may not expect
- value types being unknown to outside users and causing failures

this is in fact a full classification of cross-compatibility issues: side-effects changing, evaluation outputs changing, and evaluation inputs changing. side-effects need not be considered very much since nixlang is supposed to be *pure* and all side-effects that are not part of the store interface must already be considered incidental. evaluation outputs changing can be handled by optional lint or runtime warnings when a versioned evaluation structure passes a semantic version boundary without being annotated as an intentional behavioral leak. evaluation inputs changing is a non-issue because nix plugins and the `ExternalValue` infrastructure already make it impossible to rely on the type system being fully specified at the time an expression is written

inter-file inter-actions

by default language features **must not** be propagated across an unadorned `import` boundary to retain compatibility with existing nix code (eg nixpkgs, which will not be able to switch for quite some time). in some circumstances it is however *required* to propagate language features across imports to provide a consistent and meaningful interface, eg in the case of a hypothetical `requiredLanguageFeatures` attribute for a flake. to allow for both of these requirements to peacefully coexist we add a new primop:

```
scopedImportUsing
:: { features ? <current language features> :: AttrSetOf bool
  ## ^ language features as would be specified by `use ...;`.
  ## selecting a default-off feature is achieved by setting its key to `true`,
  ## deselecting a default-on feature is achieved by setting its key to `false`.
  ## nesting is not needed because features are identifiers. future changes to
  ## the use interface may extend the type of this set.
, newGlobals ? env: env :: AttrSetOf Any -> AttrSetOf Any
  ## ^ function to produce the new global environment. it receives the default globals
  ## set for the target expression language features (as calculated from `features` and
  ## the target `use` clause) and produces a new set.
  ## `scopedImport` behavior is recovered by setting this to `const newEnv`.
}
-> PathLike
## ^ imported path as in `scopedImport`
-> Any
## ^ import result. may be cached, most immediately using the intransparent internal
## object id of the provided features and the globals set. this mimics the behavior
## or `import` in cppnix
```

if the imported expression selects a different set of language features the features specified by `scopedImportUsing` are ignored.

`scopedImportUsing` is available in the `builtins` set and crucially, *can be replaced*. this allows a hypothetical flake implementation to replace both `scopedImportUsing` and `import` with its own versions that provide propagation behaviors that might be expected from such a library:

- importing within the same flake simply propagates the language features as-is
- importing across flake boundaries first resolves the language versions used by the imported-from flake, then applies and propagates using these features. if the imported-from flake then imports code from elsewhere this cycle repeats and can eventually restore the language features set to its original value when importing code next to the code importing the importing code

- importing out of a flake boundary (as might be possible in an impure mode) resets the propagated language feature set as if it had never been set in the first place

additionally the current language features might be made available through a builtin value `languageFeatures` by such a replacement of `scopedImportUsing`.

builtins versioning, global versions

a language feature may add or remove elements of `builtins` or the global environment. as mentioned earlier this does not pose a large hazard since evaluation is sufficiently unspecified that this must *already* be expected to happen.

interactions with eg nixpkgs lib

nixpkgs lib (and other libraries) will have to cater to the smallest common denominator when exposing library functions/constants as they do now. if we change a function to have a different prototype and a library reexports it from builtins to its own namespace the language features used by the code importing the library do not matter. to make this problem less unbearable we may want to introduce a concept of library objects and a "use library" directive like eg python `from ... import ...` that can pass language features down to the library being imported in some way.

as a first approximation it would be sufficient to encourage libraries to version their namespaces in such a way that accessing a namespace that relies on language features not present in the current evaluator will fail to evaluate (eg by providing the library itself as a plain set and each version as an attribute that (lazily) imports the specific version of the library needed to fulfill the requested version).

bad ideas for features to remove/change in the first langver

- remove url literals
- remove `with`
- remove `rec` (including `__overrides`)
- remove `let { body = ...; ... }`
- remove `or` contextual keyword, either rework or make a real keyword
- extend `listToAttrs` prototype to also accept 2-tuples instead of name-value-attrpairs
- remove `__sub` and similar overloading

Feature detection

jade: I think we might want to be able to feature detect certain features, e.g. new builtin args, which can be done without, but we would like to know if they are there.

`builtins.nixVersion` has been defanged, which means that an alternate cross impl compatible mechanism needs to be created.

Minimally thought-through proposal

`builtins.features` is an attribute set, where individual attribute names are exposed with the value true if they are implemented by a given implementation.

Attribute names are of the format:

"domainname.feature", for example, "systems.lix.somefeature".

Docs rewrite plans

Here, for now (public edit link): <https://pad.lix.systems/lix-docs-planning>