> [!TIP] Are you interested in hacking on Nix projects for a week, together with other Nix users? Do you have time at the end of August? Great, come join us at Volga Sprint!

Nix - A One Pager

Nix, the package manager, is built on and with Nix, the language. This page serves as a fast intro to most of the (small) language.

Unless otherwise specified, the word "Nix" refers only to the language below.

Please file an issue if something in here confuses you or you think something important is missing.

If you have Nix installed, you can try the examples below by running `nix repl` and entering code snippets there.

Table of Contents

Overview

Nix is:

- **purely functional**. It has no concept of sequential steps being executed, any dependency between operations is established by depending on *data* from previous operations.

  Any valid piece of Nix code is an *expression* that returns a value.

  Evaluating a Nix expression *yields a single data structure*, it does not execute a sequence of operations.

Every Nix file evaluates to a *single expression*.

- **lazy**. It will only evaluate expressions when their result is actually requested.

  For example, the builtin function `throw` causes evaluation to stop. Entering the following expression works fine however, because we never actually ask for the part of the structure that causes the `throw`.

  ```
  let attrs = { a = 15; b = builtins.throw "Oh no!"; };
  in "The value of 'a' is ${toString attrs.a}"
  ```

- **purpose-built**. Nix only exists to be the language for Nix, the package manager. While people have occasionally used it for other use-cases, it is explicitly not a general-purpose language.

Language constructs

This section describes the language constructs in Nix. It is a small language and most of these should be self-explanatory.

Primitives / literals

Nix has a handful of data types which can be represented literally in source code, similar to many other languages.

```
# numbers
42
1.72394

# strings & paths
"hello"
./some-file.json

# strings support interpolation
"Hello ${name}"

# multi-line strings (common prefix whitespace is dropped)
''
first line
second line
''

# lists (note: no commas!)
[ 1 2 3 ]

# attribute sets (field access with dot syntax)
{ a = 15; b = "something else"; }

# recursive attribute sets (fields can reference each other)
rec { a = 15; b = a * 2; }
```

Operators

Nix has several operators, most of which are unsurprising:

Syntax	Description
`+, -, *, /`	Numerical operations
`+`	String concatenation
`++`	List concatenation
`==`	Equality
`>, >=, <, <=`	Ordering comparators
`&&`	Logical `AND`
`||`	Logical `OR`
`e1 -> e2`	Logical implication (i.e. `!e1 || e2`)

Syntax	Description
`!`	Boolean negation
`set.attr`	Access attribute `attr` in attribute set `set`
`set ? attribute`	Test whether attribute set contains an attribute
`left // right`	Merge `left` & `right` attribute sets, with the right set taking precedence

`//` (merge) operator

The `//`-operator is used pervasively in Nix code. You should familiarise yourself with it, as it is likely also the least familiar one.

It merges the left and right attribute sets given to it:

```
{ a = 1; } // { b = 2; }

# yields { a = 1; b = 2; }
```

Values from the right side take precedence:

```
{ a = "left"; } // { a = "right"; }

# yields { a = "right"; }
```

The merge operator does *not* recursively merge attribute sets;

```
{ a = { b = 1; }; } // { a = { c = 2; }; }

# yields { a = { c = 2; }; }
```

Helper functions for recursive merging exist in the `lib` library.

Variable bindings

Bindings in Nix are introduced locally via `let` expressions, which make some variables available within a given scope.

For example:

```
let
  a = 15;
  b = 2;
in a * b

# yields 30
```

Variables are immutable. This means that after defining what `a` or `b` are, you can not *modify* their value in the scope in which they are available.

You can nest `let`-expressions to shadow variables.

Variables are *not* available outside of the scope of the `let` expression. There are no global variables.

Functions

All functions in Nix are anonymous lambdas. This means that they are treated just like data. Giving them names is accomplished by assigning them to variables, or setting them as values in an attribute set (more on that below).

```
# simple function
# declaration is simply the argument followed by a colon
name: "Hello ${name}"
```

Multiple arguments (currying)

Technically any Nix function can only accept **one argument**. Sometimes however, a function needs multiple arguments. This is achieved in Nix via currying, which means to create a function with one argument, that returns a function with another argument, that returns ... and so on.

For example:

```
name: age: "${name} is ${toString age} years old"
```

An additional benefit of this approach is that you can pass one parameter to a curried function, and receive back a function that you can re-use (similar to partial application):

```
let
  multiply = a: b: a * b;
  doubleIt = multiply 2; # at this point we have passed in the value for 'a' and
                         # receive back another function that still expects 'b'
in
  doubleIt 15

# yields 30
```

Multiple arguments (attribute sets)

Another way of specifying multiple arguments to a function in Nix is to make it accept an attribute set, which enables multiple other features:

```
{ name, age }: "${name} is ${toString age} years old"
```

Using this method, we gain the ability to specify default arguments (so that callers can omit them):

```
{ name, age ? 42 }: "${name} is ${toString age} years old"
```

Or in practice:

```
let greeter =  { name, age ? 42 }: "${name} is ${toString age} years old";
in greeter { name = "Slartibartfast"; }

# yields "Slartibartfast is 42 years old"
# (note: Slartibartfast is actually /significantly/ older)
```

Additionally we can introduce an ellipsis using `...`, meaning that we can accept an attribute set as our input that contains more variables than are needed for the function.

```
let greeter = { name, age, ... }: "${name} is ${toString age} years old";
    person = {
      name = "Slartibartfast";
      age = 42;
      # the 'email' attribute is not expected by the 'greeter' function ...
      email = "slartibartfast@magrath.ea";
    };
in greeter person # ... but the call works due to the ellipsis.
```

Nix also supports binding the whole set of passed in attributes to a parameter using the @ syntax:

```
let func = { name, age, ... }@args: builtins.attrNames args;
in func {
    name = "Slartibartfast";
    age = 42;
    email = "slartibartfast@magrath.ea";
}

# yields: [ "age" "email" "name" ]
```

Warning: Combining the @ syntax with default arguments can lead to surprising behaviour, as the passed attributes are bound verbatim. This means that defaulted arguments are not included in the bound attribute set:

```
({ a ? 1, b }@args: args.a) { b = 1; }
# throws: error: attribute 'a' missing

({ a ? 1, b }@args: args.a) { b = 1; a = 2; }
# => 2
```

`if ... then ... else ...`

Nix has simple conditional support. Note that `if` is an **expression** in Nix, which means that both branches must be specified.

```
if someCondition
then "it was true"
else "it was false"
```

`inherit` keyword

The `inherit` keyword is used in attribute sets or `let` bindings to "inherit" variables from the parent scope.

In short, a statement like `inherit foo;` expands to `foo = foo;`.

Consider this example:

```
let
  name = "Slartibartfast";
  # ... other variables
in {
  name = name; # set the attribute set key 'name' to the value of the 'name' var
  # ... other attributes
}
```

The `name = name;` line can be replaced with `inherit name;`:

```
let
  name = "Slartibartfast";
  # ... other variables
in {
  inherit name;
  # ... other attributes
}
```

This is often convenient, especially because inherit supports multiple variables at the same time as well as "inheritance" from other attribute sets:

```
{
  inherit name age; # equivalent to `name = name; age = age;`
  inherit (otherAttrs) email; # equivalent to `email = otherAttrs.email`;
}
```

`with` statements

The `with` statement "imports" all attributes from an attribute set into variables of the same name:

```
let attrs = { a = 15; b = 2; };
in with attrs; a + b # 'a' and 'b' become variables in the scope following 'with'
```

The scope of a `with`-"block" is the expression immediately following the semicolon, i.e.:

```
let attrs = { /* some attributes */ };
in with attrs; (/* this is the scope of the `with` */)
```

`import` / `NIX_PATH` / `<entry>`

Nix files can import each other by using the builtin `import` function and a literal path:

```
# assuming there is a file lib.nix with some useful functions
let myLib = import ./lib.nix;
in myLib.usefulFunction 42
```

The `import` function will read and evaluate the file, and return its Nix value.

Nix files often begin with a function header to pass parameters into the rest of the file, so you will often see imports of the form `import ./some-file { ... }`.

Nix has a concept of a `NIX_PATH` (similar to the standard `PATH` environment variable) which contains named aliases for file paths containing Nix expressions.

In a standard Nix installation, several channels will be present (for example `nixpkgs` or `nixos-unstable`) on the `NIX_PATH`.

`NIX_PATH` entries can be accessed using the `<entry>` syntax, which simply evaluates to their file path:

```
<nixpkgs>
# might yield something like `/home/tazjin/.nix-defexpr/channels/nixpkgs`
```

This is commonly used to import from channels:

```
let pkgs = import <nixpkgs> {};
in pkgs.something
```

`or` expressions

Nix has a keyword called `or` which can be used to access a value from an attribute set while providing a fallback to a default value.

The syntax is simple:

```nix
# Access an existing attribute
let set = { a = 42; };
in set.a or 23
```

Since the attribute a exists, this will return `42`.

```nix
# ... or fall back to a default if there is no such key
let set = { };
in set.a or 23
```

Since the attribute a does not exist, this will fall back to returning the default value `23`.

Note that `or` expressions also work for nested attribute set access.

Standard libraries

Yes, libraries, plural.

Nix has three major things that could be considered its standard library and while there's a lot of debate to be had about this point, you still need to know all three.

`builtins`

Nix comes with several functions that are baked into the language. These work regardless of which other Nix code you may or may not have imported.

Most of these functions are implemented in the Nix interpreter itself, which means that they are rather fast when compared to some of the equivalents which are implemented in Nix itself.

The Nix manual has a section listing all `builtins` and their usage.

Examples of builtins that you will commonly encounter include, but are not limited to:

- `derivation` (see Derivations)
- `toJSON` / `fromJSON`
- `toString`
- `toPath` / `fromPath`

The builtins also include several functions that have the (spooky) ability to break Nix' evaluation purity. No functions written in Nix itself can do this.

Examples of those include:

- `fetchGit` which can fetch a git-repository using the environment's default git/ssh configuration
- `fetchTarball` which can fetch & extract archives without having to specify hashes

Read through the manual linked above to get the full overview.

`pkgs.lib`

The Nix package set, commonly referred to by Nixers simply as nixpkgs, contains a child attribute set called `lib` which provides a large number of useful functions.

The canonical definition of these functions is their source code. I wrote a tool (nixdoc) in 2018 which generates manual entries for these functions, however not all of the files are included as of July 2019.

See the Nixpkgs manual entry on `lib` for the documentation.

These functions include various utilities for dealing with the data types in Nix (lists, attribute sets, strings etc.) and it is useful to at least skim through them to familiarise yourself with what is available.

```nix
{ pkgs ? import <nixpkgs> {} }:

with pkgs.lib; # bring contents pkgs.lib into scope

strings.toUpper "hello"

# yields "HELLO"
```

pkgs itself

The Nix package set itself does not just contain packages, but also many useful functions which you might run into while creating new Nix packages.

One particular subset of these that stands out are the trivial builders, which provide utilities for writing text files or shell scripts, running shell commands and capturing their output and so on.

```nix
{ pkgs ? import <nixpkgs> {} }:

pkgs.writeText "hello.txt" "Hello dear reader!"

# yields a derivation which creates a text file with the above content
```

Derivations

When a Nix expression is evaluated it may yield one or more *derivations*. Derivations describe a single build action that, when run, places one or more outputs (whether they be files or folders) in the Nix store.

The builtin function `derivation` is responsible for creating derivations at a lower level. Usually when Nix users create derivations they will use the higher-level functions such as stdenv.mkDerivation.

Please see the manual on derivations for more information, as the general build logic is out of scope for this document.

Nix Idioms

There are several idioms in Nix which are not technically part of the language specification, but will commonly be encountered in the wild.

This section is an (incomplete) list of them.

File lambdas

It is customary to start every file with a function header that receives the files dependencies, instead of importing them directly in the file.

Sticking to this pattern lets users of your code easily change out, for example, the specific version of `nixpkgs` that is used.

A common file header pattern would look like this:

```nix
{ pkgs ? import <nixpkgs> {} }:

# ... 'pkgs' is then used in the code
```

In some sense, you might consider the function header of a file to be its "API".

`callPackage`

Building on the previous pattern, there is a custom in nixpkgs of specifying the dependencies of your file explicitly instead of accepting the entire package set.

For example, a file containing build instructions for a tool that needs the standard build environment and `libsvg` might start like this:

```nix
# my-funky-program.nix
{ stdenv, libsvg }:

stdenv.mkDerivation { ... }
```

Any time a file follows this header pattern it is probably meant to be imported using a special function called `callPackage` which is part of the top-level package set (as well as certain subsets, such as `haskellPackages`).

```nix
{ pkgs ? import <nixpkgs> {} }:

let my-funky-program = pkgs.callPackage ./my-funky-program.nix {};
in # ... something happens with my-funky-program
```

The `callPackage` function looks at the expected arguments (via `builtins.functionArgs`) and passes the appropriate keys from the set in which it is defined as the values for each corresponding argument.

Overrides / Overlays

One of the most powerful features of Nix is that the representation of all build instructions as data means that they can easily be *overridden* to get a different result.

For example, assuming there is a package `someProgram` which is built without our favourite configuration flag (`--mimic-threaten-tag`) we might override it like this:

```nix
someProgram.overrideAttrs(old: {
    configureFlags = old.configureFlags or [] ++ ["--mimic-threaten-tag"];
})
```

This pattern has a variety of applications of varying complexity. The top-level package set itself can have an `overlays` argument passed to it which may add new packages to the imported set.

Note the use of the `or` operator to default to an empty list if the original flags do not include `configureFlags`. This is required in case a package does not set any flags by itself.

Since this can change in a package over time, it is useful to guard against it using `or`.

For a slightly more advanced example, assume that we want to import `<nixpkgs>` but have the modification above be reflected in the imported package set:

```nix
let
  overlay = (final: prev: {
    someProgram = prev.someProgram.overrideAttrs(old: {
      configureFlags = old.configureFlags or [] ++ ["--mimic-threaten-tag"];
    });
  });
in import <nixpkgs> { overlays = [ overlay ]; }
```

The overlay function receives two arguments, `final` and `prev`. `final` is the fixed point of the overlay's evaluation, i.e. the package set *including* the new packages and `prev` is the "original" package set.

See the Nix manual sections on overrides and on overlays for more details (note: the convention has moved away from using `self` in favor of `final`, and `prev` instead of `super`, but the documentation has not been updated to reflect this).