Nix language basics

Contents

The Nix language is designed for conveniently creating and composing *derivations* – precise descriptions of how contents of existing files are used to derive new files. It is a domain-specific, purely functional, lazily evaluated, dynamically typed programming language.

Notable uses of the Nix language

- Nixpkgs

  The largest, most up-to-date software distribution in the world, and written in the Nix language.
- NixOS

  A Linux distribution that can be configured fully declaratively and is based on Nix and Nixpkgs.

  Its underlying modular configuration system is written in the Nix language, and uses packages from Nixpkgs. The operating system environment and services it provides are configured with the Nix language.

You may quickly encounter Nix language expressions that look very complicated. As with any programming language, the required amount of Nix language code closely matches the

Skip to main content

its solution – is understood. Building software is a complex undertaking, and Nix both *exposes* and *allows managing* this complexity with the Nix language.

Yet, the Nix language itself has only few basic concepts that will be introduced in this tutorial, and which can be combined arbitrarily. What may look complicated comes not from the language, but from how it is used.

Overview

This is an introduction to **reading the Nix language**, for the purpose of following other tutorials and examples.

Using the Nix language in practice entails multiple things:

- Language: syntax and semantics
- Libraries: `builtins` and `pkgs.lib`
- Developer tools: testing, debugging, linting, formatting, …
- Generic build mechanisms: `stdenv.mkDerivation`, trivial builders, …
- Composition and configuration mechanisms: `override`, `overrideAttrs`, overlays, `callPackage`, …
- Ecosystem-specific packaging mechanisms: `buildGoModule`, `buildPythonApplication`, …
- NixOS module system: `config`, `option`, …

This tutorial only covers the most important language features, briefly discusses libraries, and at the end will direct you to reference material and resources on the other components.

What will you learn?

This tutorial should enable you to read typical Nix language code and understand its structure. Its goal is to highlight where the Nix language may differ from languages you are used to.

It therefore shows the most common and distinguishing patterns in the Nix language:

- Assigning names and accessing values
- Declaring and calling functions
- Built-in and library functions

Skip to main content

- Derivations that describe build tasks

Important

This tutorial *does not* explain all Nix language features in detail and *does not* go into specifics of syntactical rules. For instance, we skip over commonplace constructs such as `if ... then ... else ...`.

See the Nix manual for a full language reference.

What do you need?

- Familiarity with software development
- Familiarity with Unix shell, to read command line examples
- A Nix installation to run the examples

How long does it take?

- No experience with functional programming: 2 hours
- Familiar with functional programming: 1 hour
- Proficient with functional programming: 30 minutes

We recommend to run all examples. Play with them to validate your assumptions and test what you have learned. Read detailed explanations if you want to make sure you fully understand the examples.

How to run the examples?

- A piece of Nix language code is a *Nix expression*.
- Evaluating a Nix expression produces a *Nix value*.
- The content of a *Nix file* (file extension `.nix`) is a Nix expression.

Note

To *evaluate* means to transform an expression into a value according to the language rules.

Skip to main content

This tutorial contains many examples of Nix expressions. Each one is followed by the expected evaluation result.

The following example is a Nix expression adding two numbers:

Expression

```
1 + 2
```

Value

```
3
```

Interactive evaluation

Use `nix repl` to evaluate Nix expressions interactively (by typing them on the command line):

```
$ nix repl
Welcome to Nix 2.13.3. Type :? for help.

nix-repl> 1 + 2
3
```

Note

The Nix language uses lazy evaluation, and `nix repl` by default only computes values when needed.

Some examples show a fully evaluated data structure for clarity. If your output does not match the example, try prepending `:p` to the input expression.

Example:

```
nix-repl> { a.b.c = 1; }
{ a = { ... }; }

nix-repl> :p { a.b.c = 1; }
{ a = { b = { c = 1; }; }; }
```

Type `:q` to exit `nix repl`.

Skip to main content

Evaluating Nix files

Use `nix-instantiate --eval` to evaluate the expression in a Nix file.

```
$ echo 1 + 2 > file.nix
$ nix-instantiate --eval file.nix
3
```

Detailed explanation

ℹ Note

`nix-instantiate --eval` will try to read from `default.nix` if no file name is specified.

```
$ echo 1 + 2 > default.nix
$ nix-instantiate --eval
3
```

ℹ Note

The Nix language uses lazy evaluation, and `nix-instantiate` by default only computes values when needed.

Some examples show a fully evaluated data structure for clarity. If your output does not match the example, try adding the `--strict` option to `nix-instantiate`.

Example:

```
$ echo "{ a.b.c = 1; }" > file.nix
$ nix-instantiate --eval file.nix
{ a = <CODE>; }
```

```
$ echo "{ a.b.c = 1; }" > file.nix
$ nix-instantiate --eval --strict file.nix
{ a = { b = { c = 1; }; }; }
```

Skip to main content

Notes on whitespace

White space is used to delimit lexical tokens, where required. It is otherwise insignificant.

Line breaks, indentation, and additional spaces are for readers’ convenience.

The following are equivalent:

Expression

```
let
 x = 1;
 y = 2;
in x + y
```

Value

```
3
```

Expression

```
let x=1;y=2;in x+y
```

Value

```
3
```

Names and values

Values in the Nix language can be primitive data types, lists, attribute sets, and functions.

We show examples of primitive data types and lists in the context of attribute sets. Later in this section we cover special features of character strings: string interpolation, file system paths, and indented strings. We deal with functions separately.

Attribute sets and `let` expressions are used to assign names to values. Assignments are denoted by a single equal sign (`=`).

Whenever you encounter an equal sign (`=`) in Nix language code:

- On its left is the assigned name.
- On its right is the value, delimited by a semicolon (`;`).

Attribute set `{ ... }`

Skip to main content

The following example shows all primitive data types, lists, and attribute sets.

Note

If you are familiar with JSON, imagine the Nix language as *JSON with functions*.

Nix language data types *without functions* work just like their counterparts in JSON and look very similar.

Nix

```
{
  string = "hello";
  integer = 1;
  float = 3.141;
  bool = true;
  null = null;
  list = [ 1 "two" false ];
  attribute-set = {
    a = "hello";
    b = 2;
    c = 2.718;
    d = false;
  }; # comments are supported
}
```

JSON

```
{
  "string": "hello",
  "integer": 1,
  "float": 3.141,
  "bool": true,
  "null": null,
  "list": [1, "two", false],
  "object": {
    "a": "hello",
    "b": 1,
    "c": 2.718,
    "d": false
  }
}
```

Note

- Attribute names usually do not need quotes.[1]
- List elements are separated by white space.[2]

Recursive attribute set `rec { ... }`

You will sometimes see attribute sets declared with `rec` prepended. This allows access to attributes from within the set.

Example:

Expression

```
rec {
  one = 1;
  two = one + 1;
```

Skip to main content

```
  three = two + 1;
}
```

Value

```
{ one = 1; three = 3; two = 2; }
```

Note

Elements in an attribute set can be declared in any order, and are ordered on evaluation.

Counter-example:

Expression

```
{
  one = 1;
  two = one + 1;
  three = two + 1;
}
```

Value

```
error: undefined variable 'one'

       at «string»:3:9:

            2|   one = 1;
            3|   two = one + 1;
             |         ^
            4|   three = two + 1;
```

`let ... in ...`

Also known as “`let` expression” or “`let` binding”

`let` expressions allow assigning names to values for repeated use.

Example:

Expression

```
let
  a = 1;
in
a + a
```

Value

Skip to main content

Detailed explanation

Names can be assigned in any order, and expressions on the right of the assignment (`=`) can refer to other assigned names.

Example:

Expression

```
let
  b = a + 1;
  a = 1;
in
a + b
```

Value

```
3
```

Detailed explanation

Only expressions within the `let` expression itself can access the newly declared names. We say: the bindings have local scope.

Counter-example:

Expression

```
{
  a = let x = 1; in x;
  b = x;
}
```

Value

```
error: undefined variable 'x'

       at «string»:3:7:

            2|   a = let x = 1; in x;
            3|   b = x;
             |       ^
            4| }
```

Attribute access

Attributes in a set are accessed with a dot (`.`) and the attribute name.

Example:

Skip to main content

Expression

```
let
  attrset = { x = 1; };
in
attrset.x
```

Value

```
1
```

Accessing nested attributes works the same way.

Example:

Expression

```
let
  attrset = { a = { b = { c = 1; }; }; };
in
attrset.a.b.c
```

Value

```
1
```

The dot (`.`) notation can also be used for assigning attributes.

Example:

Expression

```
{ a.b.c = 1; }
```

Value

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

`with ...; ...`

The `with` expression allows access to attributes without repeatedly referencing their attribute set.

Example:

Expression

```
let
  a = {
    x = 1;
    y = 2;
    z = 3;
  };
```

Skip to main content

```
in
with a; [ x y z ]
```

Value

```
[ 1 2 3 ]
```

The expression

```
with a; [ x y z ]
```

is equivalent to

```
[ a.x a.y a.z ]
```

Attributes made available through `with` are only in scope of the expression following the semicolon (`;`).

Counter-example:

Expression

```
let
  a = {
    x = 1;
    y = 2;
    z = 3;
  };
in
{
  b = with a; [ x y z ];
  c = x;
}
```

Value

```
error: undefined variable 'x'

       at «string»:10:7:

            9|   b = with a; [ x y z ];
           10|   c = x;
             |       ^
           11| }
```

`inherit ...`

`inherit` is shorthand for assigning the value of a name from an existing scope to the same

Skip to main content

times.

Example:

Expression

```
let
  x = 1;
  y = 2;
in
{
  inherit x y;
}
```

Value

```
{ x = 1; y = 2; }
```

The fragment

```
inherit x y;
```

is equivalent to

```
x = x; y = y;
```

It is also possible to `inherit` names from a specific attribute set with parentheses (`inherit (...) ...`).

Example:

Expression

```
let
  a = { x = 1; y = 2; };
in
{
  inherit (a) x y;
}
```

Value

```
{ x = 1; y = 2; }
```

The fragment

```
inherit (a) x y;
```

Skip to main content

```
x = a.x; y = a.y;
```

`inherit` also works inside `let` expressions.

Example:

Expression

```
let
  inherit ({ x = 1; y = 2; }) x y;
in [ x y ]
```

Value

```
[ 1 2 ]
```

Detailed explanation

String interpolation `${ ... }`

Previously known as “antiquotation”.

The value of a Nix expression can be inserted into a character string with the dollar-sign and braces (`${ }`).

Example:

Expression

```
let
  name = "Nix";
in
"hello ${name}"
```

Value

```
"hello Nix"
```

Only character strings or values that can be represented as a character string are allowed.

Counter-example:

Expression

```
let
  x = 1;
in
"${x} + ${x} = ${x + x}"
```

Skip to main content

Value

```
error: cannot coerce an integer to a string

       at «string»:4:2:

            3| in
            4| "${x} + ${x} = ${x + x}"
             |  ^
            5|
```

Interpolated expressions can be arbitrarily nested.

(This can become hard to read, and we recommend to avoid it in practice.)

Example:

Expression

```
let
  a = "no";
in
"${a + " ${a + " ${a}"}"}"
```

Value

```
"no no no"
```

Detailed explanation

⚠ Warning

You may encounter strings that use the dollar sign (`$`) before an assigned name, but no braces (`{ }`):

These are *not* interpolated strings, but usually denote variables in a shell script.

In such cases, the use of names from the surrounding Nix expression is a coincidence.

Example:

Expression

```
let
  out = "Nix";
in
"echo ${out} > $out"
```

Value

```
"echo Nix > $out"
```

Skip to main content

File system paths

The Nix language offers convenience syntax for file system paths.

Absolute paths always start with a slash (`/`).

Example:

Expression

```
/absolute/path
```

Value

```
/absolute/path
```

Paths are relative when they contain at least one slash (`/`) but do not start with one. They evaluate to the path relative to the file containing the expression.

The following examples assume the containing Nix file is in `/current/directory` (or `nix repl` is run in `/current/directory`).

Example:

Expression

```
./relative
```

Value

```
/current/directory/relative
```

Example:

Expression

```
relative/path
```

Value

```
/current/directory/relative/path
```

One dot (`.`) denotes the current directory within the given path.

You will often see the following expression, which specifies a Nix file’s directory.

Example:

Expression

```
./.
```

Skip to main content

Value

```
/current/directory
```

Detailed explanation

Two dots (`..`) denote the parent directory.

Example:

Expression

```
../.
```

Value

```
/current
```

Note

Paths can be used in interpolated expressions – an impure operation we will cover in detail in a later section.

Lookup paths

Also known as “angle bracket syntax”.

Example:

Expression

```
<nixpkgs>
```

Value

```
/nix/var/nix/profiles/per-user/root/channels/nixpkgs
```

The value of a lookup path is a file system path that depends on the value of `builtins.nixPath`.

In practice, `<nixpkgs>` points to the file system path of some revision of Nixpkgs.

For example, `<nixpkgs/lib>` points to the subdirectory `lib` of that file system path:

Expression

```
<nixpkgs/lib>
```

Skip to main content

Value

```
/nix/var/nix/profiles/per-user/root/channels/nixpkgs/lib
```

While you will encounter many such examples, we recommend to avoid lookup paths in production code, as they are impurities which are not reproducible.

Indented strings

Also known as “multi-line strings”.

The Nix language offers convenience syntax for character strings which span multiple lines that have common indentation.

Indented strings are denoted by *double single quotes* (`'' ''`).

Example:

Expression

```
''
multi
line
string
''
```

Value

```
"multi\nline\nstring\n"
```

Equal amounts of prepended white space are trimmed from the result.

Example:

Expression

```
''
  one
   two
    three
''
```

Value

```
"one\n two\n  three\n"
```

Functions

Skip to main content

A function always takes exactly one argument. Argument and function body are separated by a colon (`:`).

Wherever you find a colon (`:`) in Nix language code:

- On its left is the function argument
- On its right is the function body.

Function arguments are the third way, apart from attribute sets and `let` expressions, to assign names to values. Notably, values are not known in advance: the names are placeholders that are filled when calling a function.

Function declarations in the Nix language can appear in different forms. Each of them is explained in the following, and here is an overview:

- Single argument

  ```
  x: x + 1
  ```

  - Multiple arguments via nesting

    ```
    x: y: x + y
    ```

- Attribute set argument

  ```
  { a, b }: a + b
  ```

  - With default attributes

    ```
    { a, b ? 0 }: a + b
    ```

  - With additional attributes allowed

    ```
    { a, b, ...}: a + b
    ```

- Named attribute set argument

  ```
  args@{ a, b, ... }: a + b + args.c
  ```

Skip to main content

```
{ a, b, ... }@args: a + b + args.c
```

Functions have no names. We say they are anonymous, and call such a function a *lambda*.[3]

Example:

Expression

```
x: x + 1
```

Value

```
<LAMBDA>
```

The `<LAMBDA>` indicates the resulting value is an anonymous function.

As with any other value, functions can be assigned to a name.

Example:

Expression

```
let
  f = x: x + 1;
in f
```

Value

```
<LAMBDA>
```

Calling functions

Also known as “function application”.

Calling a function with an argument means writing the argument after the function.

Example:

Expression

```
let
  f = x: x + 1;
in f 1
```

Value

```
2
```

Example:

Skip to main content

Expression

```
let
  f = x: x.a;
in
f { a = 1; }
```

Value

```
1
```

The above example calls `f` on a literal attribute set. One can also pass arguments by name.

Example:

Expression

```
let
  f = x: x.a;
  v = { a = 1; };
in
f v
```

Value

```
1
```

Since function and argument are separated by white space, sometimes parentheses (`( )`) are required to achieve the desired result.

Example:

Expression

```
(x: x + 1) 1
```

Value

```
2
```

Detailed explanation

Example:

List elements are also separated by white space, therefore the following are different:

Expression

```
let
 f = x: x + 1;
 a = 1;
in [ (f a) ]
```

Skip to main content

Value

```
[ 2 ]
```

Expression

```
let
 f = x: x + 1;
 a = 1;
in [ f a ]
```

Value

```
[ <LAMBDA> 1 ]
```

The first example reads: apply `f` to `a`, and put the result in a list. The resulting list has one element.

The second example reads: put `f` and `a` in a list. The resulting list has two elements.

Multiple arguments

Also known as “curried functions”.

Nix functions take exactly one argument. Multiple arguments can be handled by nesting functions.

Such a nested function can be used like a function that takes multiple arguments, but offers additional flexibility.

Example:

Expression

```
x: y: x + y
```

Value

```
<LAMBDA>
```

The above function is equivalent to

Expression

```
x: (y: x + y)
```

Value

```
<LAMBDA>
```

This function takes one argument and returns another function `y: x + y` with `x` set to the

Skip to main content

Example:

Expression

```
let
  f = x: y: x + y;
in
f 1
```

Value

```
<LAMBDA>
```

Applying the function which results from `f 1` to another argument yields the inner body `x + y` (with `x` set to `1` and `y` set to the other argument), which can now be fully evaluated.

Expression

```
let
  f = x: y: x + y;
in
f 1 2
```

Value

```
3
```

Attribute set argument

Also known as “keyword arguments” or “destructuring” .

Nix functions can be declared to require an attribute set with specific structure as argument.

This is denoted by listing the expected attribute names separated by commas (`,`) and enclosed in braces (`{ }`).

Example:

Expression

```
{a, b}: a + b
```

Value

```
<LAMBDA>
```

The argument defines the exact attributes that have to be in that set. Leaving out or passing additional attributes is an error.

Example:

Skip to main content

Expression

```
let
  f = {a, b}: a + b;
in
f { a = 1; b = 2; }
```

Value

```
3
```

Counter-example:

Expression

```
let
  f = {a, b}: a + b;
in
f { a = 1; b = 2; c = 3; }
```

Value

```
error: 'f' at (string):2:7 called with unexpected argument 'c'

       at «string»:4:1:

            3| in
            4| f { a = 1; b = 2; c = 3; }
             | ^
            5|
```

Default values

Also known as “default arguments”.

Destructured arguments can have default values for attributes.

This is denoted by separating the attribute name and its default value with a question mark (`?`).

Attributes in the argument are not required if they have a default value.

Example:

Expression

```
let
  f = {a, b ? 0}: a + b;
in
f { a = 1; }
```

Value

Skip to main content

Example:

Expression

```
let
  f = {a ? 0, b ? 0}: a + b;
in
f { } # empty attribute set
```

Value

```
0
```

Additional attributes

Additional attributes are allowed with an ellipsis (`...`):

```
{a, b, ...}: a + b
```

Unlike in the previous counter-example, passing an argument that contains additional attributes is not an error.

Example:

Expression

```
let
  f = {a, b, ...}: a + b;
in
f { a = 1; b = 2; c = 3; }
```

Value

```
3
```

Named attribute set argument

Also known as “@ pattern”, “@ syntax”, or “‘at’ syntax”.

An attribute set argument can be given a name to be accessible as a whole.

This is denoted by prepending or appending the name to the attribute set argument, separated by the at sign (`@`).

Example:

Skip to main content

Expression

```
{a, b, ...}@args: a + b + args.c
```

Value

```
<LAMBDA>
```

or

Expression

```
args@{a, b, ...}: a + b + args.c
```

Value

```
<LAMBDA>
```

Example:

Expression

```
let
  f = {a, b, ...}@args: a + b + args.c;
in
f { a = 1; b = 2; c = 3; }
```

Value

```
6
```

Function libraries

In addition to the built-in operators (`+`, `==`, `&&`, etc.), there are two widely used libraries that *together* can be considered standard for the Nix language. You need to know about both to understand and navigate Nix language code.

We recommend to at least skim them to familiarise yourself with what is available.

`builtins`

Also known as “primitive operations” or “primops”.

Nix comes with many functions that are built into the language. They are implemented in C++ as part of the Nix language interpreter.

Skip to main content

Note

The Nix manual lists all Built-in Functions, and shows how to use them.

These functions are available under the `builtins` constant.

Example:

Expression

```
builtins.toString
```

Value

```
<PRIMOP>
```

`import`

Most built-in functions are only accessible through `builtins`. A notable exception is `import`, which is also available at the top level.

`import` takes a path to a Nix file, reads it to evaluate the contained Nix expression, and returns the resulting value. If the path points to a directory, the file `default.nix` in that directory is used instead.

Example:

```
$ echo 1 + 2 > file.nix
```

Expression

```
import ./file.nix
```

Value

```
3
```

Detailed explanation

Since a Nix file can contain any Nix expression, `import`ed functions can be applied to arguments immediately.

That is, whenever you find additional tokens after a call to `import`, the value it returns should be a function, and anything that follows are arguments to that function.

Skip to main content

Example:

```
$ echo "x: x + 1" > file.nix
```

Expression

```
import ./file.nix 1
```

Value

```
2
```

Detailed explanation

`pkgs.lib`

The `nixpkgs` repository contains an attribute set called `lib`, which provides a large number of useful functions. They are implemented in the Nix language, as opposed to `builtins`, which are part of the language itself.

> **Note**
>
> The Nixpkgs manual lists all Nixpkgs library functions.

These functions are usually accessed through `pkgs.lib`, as the Nixpkgs attribute set is given the name `pkgs` by convention.

Example:

Expression

```
let
  pkgs = import <nixpkgs> {};
in
pkgs.lib.strings.toUpper "lookup paths considered harmful"
```

Value

```
LOOKUP PATHS CONSIDERED HARMFUL
```

Detailed explanation

For historical reasons, some of the functions in `pkgs.lib` are equivalent to `builtins` of the same name.

Skip to main content

Impurities

So far we have only covered what we call *pure expressions*: declaring data and transforming it with functions.

In practice, describing derivations – the Nix language's defining feature, which enables functional programming with the file system – requires observing the outside world. We will discuss derivations later in the tutorial.

There is only one impurity in the Nix language that is relevant here: reading files from the file system as *build inputs*.

Build inputs are files that derivations refer to in order to describe how to derive new files. When run, a derivation will only have access to explicitly declared build inputs.

The only way to specify build inputs in the Nix language is explicitly with:

- File system paths
- Dedicated functions

Nix and the Nix language refer to files by their content hash. If file contents are not known in advance, it's unavoidable to read files during expression evaluation.

> **Note**
>
> Nix supports other types of impure expressions, such as lookup paths or the constant `builtins.currentSystem`. We do not cover those here in more detail, as they do not matter for how the Nix language works in principle, and because they are discouraged for the very reason of breaking reproducibility.

Paths

Whenever a file system path is used in string interpolation, the contents of that file are copied to a special location in the file system, the *Nix store*, as a side effect.

The evaluated string then contains the Nix store path assigned to that file.

Example:

```
$ echo 123 > data
```

Skip to main content

Expression

```
"${./data}"
```

Value

```
"/nix/store/h1qj5h5n05b5dl5q4nldrqq8mdg7dhqk-data"
```

Detailed explanation

For directories the same thing happens: The entire directory (including nested files and directories) is copied to the Nix store, and the evaluated string becomes the Nix store path of the directory.

Fetchers

Files to be used as build inputs do not have to come from the file system.

The Nix language provides built-in impure functions to fetch files over the network during evaluation:

- `builtins.fetchurl`
- `builtins.fetchTarball`
- `builtins.fetchGit`
- `builtins.fetchClosure`

These functions evaluate to a file system path in the Nix store.

Example:

Expression

```
builtins.fetchurl "https://github.com/NixOS/nix/archive/7c3ab5751568a0bc63430b33a51
```

Value

```
"/nix/store/7dhgs330clj36384akg86140fqkgh8zf-7c3ab5751568a0bc63430b33a5169c5e4784a0
```

Some of them add extra convenience, such as automatically unpacking archives.

Example:

Expression

```
builtins.fetchTarball "https://github.com/NixOS/nix/archive/7c3ab5751568a0bc63430b3
```

Skip to main content

Value

```
"/nix/store/d59llm96vgis5fy231x6m7nrijs0ww36-source"
```

> **Note**
>
> The Nixpkgs manual on Fetchers lists numerous additional library functions to fetch files over the network.

It is an error if the network request fails.

Derivations

Derivations are at the core of both Nix and the Nix language:

- The Nix language is used to describe derivations.
- Nix runs derivations to produce *build results*.
- Build results can in turn be used as inputs for other derivations.

The Nix language primitive to declare a derivation is the built-in impure function `derivation`.

It is usually wrapped by the Nixpkgs build mechanism `stdenv.mkDerivation`, which hides much of the complexity involved in non-trivial build procedures.

> **Note**
>
> You will probably never encounter `derivation` in practice.

Whenever you encounter `mkDerivation`, it denotes something that Nix will eventually *build*.

Example: a package using **`mkDerivation`**

The evaluation result of `derivation` (and `mkDerivation`) is an attribute set with a certain structure and a special property: It can be used in string interpolation, and in that case evaluates to the Nix store path of its build result.

Example:

Expression

```
let
```

Skip to main content

```
in "${pkgs.nix}"
```

Value

```
"/nix/store/sv2srrjddrp2isghmrla8s6lazbzmikd-nix-2.11.0"
```

Note

Your output may differ. It may produce a different hash or even a different package version.

A derivation’s output path is fully determined by its inputs, which in this case come from *some* version of Nixpkgs.

This is why we recommend to avoid lookup paths to ensure predictable outcomes, except in examples intended for illustration only.

Detailed explanation

String interpolation on derivations is used to refer to their build results as file system paths when declaring new derivations.

This allows constructing arbitrarily complex compositions of derivations with the Nix language.

Worked examples

So far we have seen artificial examples illustrating the various constructs in the Nix language.

You should now be able to read Nix language code for simple packages and configurations, and come up with similar explanations of the following practical examples.

Note

The goal of the following exercises is not to understand what the code means or how it works, but how it is structured in terms of functions, attribute sets, and other Nix language data types.

Skip to main content

Shell environment

```
{ pkgs ? import <nixpkgs> {} }:
let
  message = "hello world";
in
pkgs.mkShellNoCC {
  packages = with pkgs; [ cowsay ];
  shellHook = ''
    cowsay ${message}
  '';
}
```

This example declares a shell environment (which runs the `shellHook` on initialization).

Explanation:

- This expression is a function that takes an attribute set as an argument.
- If the argument has the attribute `pkgs`, it will be used in the function body. Otherwise, by default, import the Nix expression in the file found on the lookup path `<nixpkgs>` (which is a function in this case), call the function with an empty attribute set, and use the resulting value.
- The name `message` is bound to the string value `"hello world"`.
- The attribute `mkShellNoCC` of the `pkgs` set is a function that is passed an attribute set as argument. Its return value is also the result of the outer function.
- The attribute set passed to `mkShellNoCC` has the attributes `packages` (set to a list with one element: the `cowsay` attribute from `pkgs`) and `shellHook` (set to an indented string).
- The indented string contains an interpolated expression, which will expand the value of `message` to yield `"hello world"`.

NixOS configuration

```
{ config, pkgs, ... }: {

  imports = [ ./hardware-configuration.nix ];

  environment.systemPackages = with pkgs; [ git ];

  # ...

}
```

Skip to main content

This example is (part of) a NixOS configuration.

Explanation:

- This expression is a function that takes an attribute set as an argument. It returns an attribute set.
- The argument must at least have the attributes `config` and `pkgs`, and may have more attributes.
- The returned attribute set contains the attributes `imports` and `environment`.
- `imports` is a list with one element: a path to a file next to this Nix file, called `hardware-configuration.nix`.

  > **Note**
  >
  > `imports` is not the impure built-in `import`, but a regular attribute name!

- `environment` is itself an attribute set with one attribute `systemPackages`, which will evaluate to a list with one element: the `git` attribute from the `pkgs` set.
- The `config` argument is not (shown to be) used.

Package

```
{ lib, stdenv, fetchurl }:

stdenv.mkDerivation rec {

  pname = "hello";

  version = "2.12";

  src = fetchurl {
    url = "mirror://gnu/${pname}/${pname}-${version}.tar.gz";
    sha256 = "1ayhp9v4m4rdhjmnl2bq3cibrbqqkgjbl3s7yk2nhlh8vj3ay16g";
  };

  meta = with lib; {
    license = licenses.gpl3Plus;
  };

}
```

This example is a (simplified) package declaration from Nixpkgs.

Skip to main content

- This expression is a function that takes an attribute set which must have exactly the attributes `lib`, `stdenv`, and `fetchurl`.
- It returns the result of evaluating the function `mkDerivation`, which is an attribute of `stdenv`, applied to a recursive set.
- The recursive set passed to `mkDerivation` uses its own `pname` and `version` attributes in the argument to the function `fetchurl`. `fetchurl` itself comes from the outer function’s arguments.
- The `meta` attribute is itself an attribute set, where the `license` attribute has the value that was assigned to the nested attribute `lib.licenses.gpl3Plus`.

References

- Nix manual: Nix language
- Nix manual: String interpolation
- Nix manual: Built-in Functions
- Nix manual: **`nix repl`**
- Nixpkgs manual: Functions reference
- Nixpkgs manual: Fetchers

Next steps

Get things done

- Declarative shell environments with shell.nix – create reproducible shell environments from a Nix file
- Packaging existing software with Nix – make more software available through Nix

If you want to take a longer break from learning Nix, you can remove unused build results from the Nix store with:

```
$ nix-collect-garbage
```

Skip to main content

Learn more

If you worked through the examples, you will have noticed that reading the Nix language reveals the structure of the code, but does not necessarily tell what the code actually means.

Often it is not possible to determine from the code at hand

- the data type of a named value or function argument.
- the data type a called function accepts for its argument.
- which attributes are present in a given attribute set.

Example:

```
{ x, y, z }: (x y) z.a
```

How do we know…

- that `x` will be a function that, given an argument, returns a function?
- that, given `x` is a function, `y` will be an appropriate argument to `x`?
- that, given `(x y)` is a function, `z.a` will be an appropriate argument to `(x y)`?
- that `z` will be an attribute set at all?
- that, given `z` is an attribute set, it will have an attribute `a`?
- which data type `y` and `z.a` will be?
- the data type of the end result?

And how does the caller of this function know that it requires an attribute set with attributes `x`, `y`, `z`?

Answering such questions requires knowing the context in which a given expression is supposed to be used.

The Nix ecosystem and code style is driven by conventions. Most names you will encounter in Nix language code come from Nixpkgs:

- Nix Pills - a detailed explanation of derivations and how Nixpkgs is constructed from first principles

Nixpkgs provides generic build mechanisms that are widely used:

- **`stdenv`** - most importantly `mkDerivation`

Skip to main content

Packages from Nixpkgs can be modified through multiple mechanisms:

- overrides – specifically `override` and `overrideAttrs` to modify single packages
- overlays – to produce a custom variant of Nixpkgs with individually modified packages

Different language ecosystems and frameworks have different requirements to accommodating them into Nixpkgs:

- Languages and frameworks lists tools provided by Nixpkgs to build language- or framework-specific packages with Nix.

The NixOS Linux distribution has a modular configuration system that imposes its own conventions:

- NixOS modules shows how NixOS configurations are organized.

[1] Details: Nix manual - attribute set

[2] Details: Nix manual - list

[3] The term *lambda* is a shorthand for lambda abstraction in the lambda calculus.