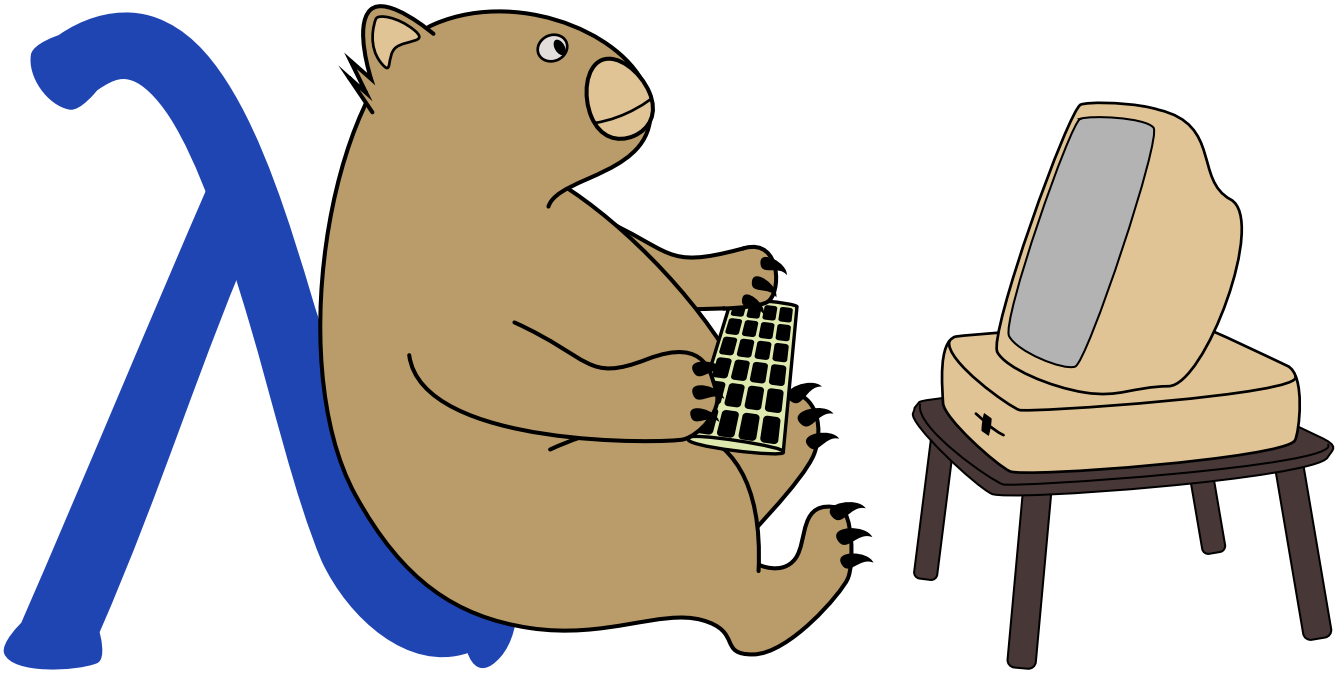
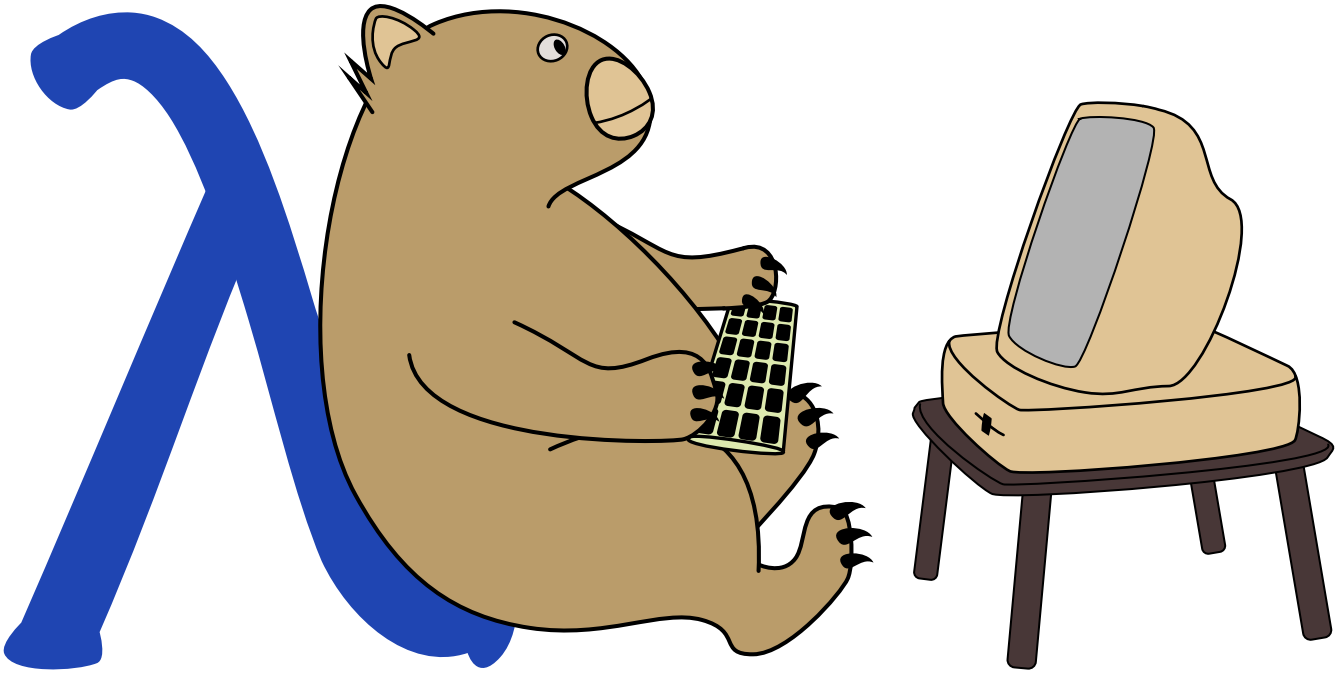


Wombat's Book of Nix

Amy de Buitléir

Table of Contents



This book is available online and as a downloadable PDF.

Last updated 2026-02-03 at 15:44:03 GMT.

Acknowledgments

I would like to thank the patient people on the NixOS Discourse Forum who answered my many questions, especially `cdepillabout`, `FedericoSchonborn`, `tejing` and `smkuehnhold`. Any mistakes in this book are my own, however.

I would also like to thank Luca Bruno (aka Lethalman). Although I have never interacted with them personally, I learned a great deal from their series of tutorials called Nix pills.

Chapter 1. Introduction

1.1. Why Nix?

If you've opened this PDF, you already have your own motivation for learning Nix. Here's how it helps me. As a researcher, I tend to work on a series of short-term projects, mostly demos and prototypes. For each one, I typically develop some software using a compiler, often with some open source libraries. Often I use other tools to analyse data or generate documentation, for example.

Problems would arise when handing off the project to colleagues; they would report errors when trying to build or run the project. Belatedly I would realise that my code relies on a library that they need to install. Or perhaps they had installed the library, but the version they're using is incompatible.

Using containers helped with the problem. However, I didn't want to *develop* in a container. I did all my development in my nice, familiar, environment with my custom aliases and shell prompt. and *then* I containerised the software. This added step was annoying for me, and if my colleague wanted to do some additional development, they would probably extract all of the source code from the container first anyway. Containers are great, but this isn't the ideal use case for them.

Nix allows me to work in my custom environment, but forces me to specify any dependencies. It automatically tracks the version of each dependency so that it can replicate the environment wherever and whenever it's needed.

1.2. Why *flakes*?

Flakes are labeled as an experimental feature, so it might seem safer to avoid them. However, they have been in use for years, and there is widespread adoption, so the aren't going away any time soon. Flakes are easier to understand, and offer more features than the traditional Nix approach. After weighing the pros and cons, I feel it's better to learn and use flakes; and this seems to be the general consensus.

1.3. Prerequisites

To follow along with the examples in this book, you will need access to a computer or (virtual machine) with Nix (or NixOS) installed and *flakes enabled*.

I recommend the installer from zero-to-nix.com. This installer automatically enables flakes.

More documentation (and another installer) available at nixos.org.

To *enable flakes on an existing Nix or NixOS installation*, see the instructions in the NixOS wiki.

There are hyphenated and un-hyphenated versions of many Nix commands. For example, `nix-shell` and `nix shell` are two different commands. Don't confuse them!

Generally speaking, the un-hyphenated versions are for working directly with

> flakes, while the hyphenated versions are for everything else.

1.4. See an error? Have a suggestion? Or want more?

If notice an error in this book, have a suggestion on how to improve it, or you're interested in an area that isn't covered, feel free to open an issue.

Chapter 2. The Nix language

2.1. Introducing the Nix language

Nix and NixOS use a functional programming language called *Nix* to specify how to build and install software, and how to configure system, user, and project-specific environments. (Yes, “Nix” is the name of both the package manager and the language it uses.)

Nix is a *functional* language. In a *procedural* language such as C or Java, the focus is on writing a series of *steps* (statements) to achieve a desired result. By contrast, in a functional language the focus is on *defining* the desired result.

In the case of Nix, the desired result is usually a *derivation*: a software package that has been built and made available for use. The Nix language has been designed for that purpose, and thus has some features you don’t typically find in general-purpose languages.

2.2. Data types

2.2.1. Strings

Strings are enclosed by double quotes (`"`), or *two* single quotes (`'`).

```
"Hello, world!"
```

```
''This string contains "double quotes"''
```

They can span multiple lines.

```
''Old pond
A frog jumps in
The sound of water
          -- Basho''
```

2.2.2. Integers

```
7
256
```

2.2.3. Floating point numbers

```
3.14
```

```
6.022e23
```

2.2.4. Boolean

The Boolean values in Nix are `true` and `false`.

2.2.5. Paths

File paths play an important role in building software, so Nix has a special data type for them. Paths may be absolute (e.g. `/bin/sh`) or relative (e.g. `./data/file1.csv`). Note that paths are not enclosed in quotation marks; they are not strings!

Enclosing a path in angle brackets, e.g. <nixpkgs> causes the directories listed in the environment variable NIX_PATH to be searched for the given file or directory name. These are called *lookup paths*.

2.2.6. Lists

List elements are enclosed in square brackets and separated by spaces (not commas). The elements need not be of the same type.

```
[ "apple" 123 ./build.sh false ]
```

Lists can be empty.

```
[]
```

List elements can be of any type, and can even be lists themselves.

```
[ [ 1 2 ] [ 3 4 ] ]
```

2.2.7. Attribute sets

Attribute sets associate keys with values. They are enclosed in curly brackets, and the associations are terminated by semi-colons. Note that the final semi-colon before the closing bracket is required.

```
{ name = "Professor Paws"; age = 10; species = "cat"; }
```

The keys of an attribute set must be strings. When the key is not a valid identifier, it must be enclosed in quotation marks.

```
{ abc = true; "123" = false; }
```

Attribute sets can be empty.

```
{}
```

Values of attribute sets can be of any type, and can even be attribute sets themselves.

```
{ name = { first = "Professor"; last = "Paws"; }; age = 10; species = "cat"; }
```

In Section 2.11.4, “Recursive attribute sets” you will be introduced to a special type of attribute set.

In some Nix documentation, and in many articles about Nix, attribute sets are simply called "sets".

2.2.8. Functions

We’ll learn how to write functions in Section 2.12, “Functions”. For now, note that functions are "first-class values", meaning that they can be treated like any other data type. For example, a function can be assigned to a variable, appear as an element in a list, be associated with a key in an attribute set, or be passed as input to another function.

```
[ "apple" 123 ./build.sh false (x: x*x) ]
{ name = "Professor Paws"; age = 10; species = "cat"; formula = (x: x*2); }
```

2.2.9. Derivations

A derivation is an attribute set, but one which is a recipe for producing a Nix package. We’ll learn how to write derivations in Section 2.14, “Derivations”. For now, note that derivations are "first-class values", meaning that they can be treated like any other data type. For example, a derivation can be assigned to a variable, appear as an element in a list, be associated with a key in an attribute set, or be passed as input to a function.

2.3. Stop reading this chapter!

When I first began using Nix, it seemed logical to start by learning the Nix language. However, after following an in-depth tutorial, I found that I didn’t know how to do anything useful with the language! It wasn’t until later that I understood what I was missing: a guide to the most useful library functions.

When working with Nix or NixOS, it’s very rare that you’ll want to write something from scratch. Instead, you’ll use one of the many library functions that make things easier and shield you from the underlying complexity. Many of these functions are language-specific, and the documentation for them may be inadequate. Often the easiest (or only) way to learn to use them is to find an example that does something similar to what you want, and then modify the function parameters to suit your needs.

At this point you've learned enough of the Nix language to do the majority of common Nix tasks. So when I say "Stop reading this chapter!", I'm only half-joking. Instead I suggest that you *skim* the rest of this chapter, paying special attention to anything marked with ❗. Then move on to the following chapters where you will learn how to develop and package software using Nix. Afterward, come back to this chapter and read it in more detail.

While writing this book, I anticipated that readers would want to skip around, alternating between pure learning and learning-by-doing. I've tried to structure the book to support that; providing extensive cross-references to earlier and later sections, and sometimes repeating information from earlier chapters that you might have skipped.

2.4. The Nix REPL

The Nix REPL (REPL is an acronym for Read-Eval-Print-Loop) is an interactive environment for evaluating and debugging Nix code. It's also a good place to begin learning Nix. Enter it using the command `nix repl`. Within the REPL, type `:?` to see a list of available commands.

```
$ nix repl
Welcome to Nix 2.18.1. Type :? for help.

nix-repl> :?
The following commands are available:

  <expr>                        Evaluate and print expression
  <x> = <expr>                  Bind expression to variable
  :a, :add <expr>               Add attributes from resulting set to scope
  :b <expr>                     Build a derivation
  :bl <expr>                    Build a derivation, creating GC roots in the
                                working directory
  :e, :edit <expr>              Open package or function in $EDITOR
  :i <expr>                     Build derivation, then install result into
                                current profile
  :l, :load <path>              Load Nix expression and add it to scope
  :lf, :load-flake <ref>        Load Nix flake and add it to scope
  :p, :print <expr>             Evaluate and print expression recursively
  :q, :quit                     Exit nix-repl
  :r, :reload                   Reload all files
  :sh <expr>                    Build dependencies of derivation, then start
                                nix-shell
  :t <expr>                     Describe result of evaluation
  :u <expr>                     Build derivation, then start nix-shell
  :doc <expr>                   Show documentation of a builtin function
  :log <expr>                   Show logs for a derivation
  :te, :trace-enable [bool]     Enable, disable or toggle showing traces for
                                errors
  :?, :help                     Brings up this help menu
```

A command that is useful to beginners is `:t`, which tells you the type of an expression.

Example

```
nix-repl> :t { abc = true; "123" = false; }
a set

nix-repl> f = x: y: x * y

nix-repl> :t f
a function
```

Note that the command to exit the REPL is `:q` (or `:quit` if you prefer).

2.5. Variables

2.5.1. Assignment

You can declare variables in Nix and assign values to them.

Example

```
nix-repl> a = 7

nix-repl> b = 3

nix-repl> a - b
4
```

The spaces before and after operators aren't always required. However, you can get unexpected results when you omit them, as shown in the following example. Nix allows hyphens (`-`) in variable names, so `a-b` is interpreted as the name of a variable rather than a subtraction operation.

Example

```
nix-repl> a-b
error: undefined variable 'a-b'

       at «string»:1:1:

            1| a-b
             | ^
```

2.5.2. Immutability

In Nix, values are *immutable*; once you assign a value to a variable, you cannot change it. You can, however, create a new variable with the same name, but in a different scope. Don't worry if you don't completely understand the previous sentence; we will see some examples in Section 2.12,

"Functions", Section 2.16, "Let expressions", and Section 2.17, "With expressions".

In the Nix REPL, it may seem like the values of variables can be changed, in *apparent* contradiction to the previous paragraph. In truth, the REPL works some behind-the-scenes "magic", effectively creating a new nested scope with each assignment. This makes it much easier to experiment in the REPL.

Example

```
nix-repl> x = 1

nix-repl> x
1

nix-repl> x = x + 1 # creates a new variable called "x"; the original
is no longer in scope

nix-repl> x
2
```

2.6. Numeric operations

2.6.1. Arithmetic operators

The usual arithmetic operators are provided.

Example

```
nix-repl> 1 + 2   # addition
3

nix-repl> 5 - 3   # subtraction
2

nix-repl> 3 * 4   # multiplication
12

nix-repl> 6 / 2   # division
3

nix-repl> -1      # negation
-1
```

As mentioned in Section 2.5, "Variables", you can get unexpected results when you omit spaces around operators. Consider the following example.

Example

```
nix-repl> 6/2
/home/amy/codeberg/nix-book/6/2
```

What happened? Let's use the `:t` command to find out the type of the expression.

Example

```
nix-repl> :t 6/2
a path
```

If an expression can be interpreted as a path, Nix will do so. This is useful, because paths are *far* more commonly used in Nix expressions than arithmetic operators. In this case, Nix interpreted `6/2` as a relative path from the current directory, which in the above example was `/home/amy/codeberg/nix-book`.

Adding a space after the `/` operator produces the expected result.

Example

```
nix-repl> 6/ 2
3
```

To avoid surprises and improve readability, I prefer to use spaces before and after all operators.

2.6.2. Floating-point calculations

Numbers without a decimal point are assumed to be integers. To ensure that a number is interpreted as a floating-point value, add a decimal point.

Example

```
nix-repl> :t 5
an integer

nix-repl> :t 5.0
a float

nix-repl> :t 5.
a float
```

In the example below, the first expression results in integer division (rounding down), while the second produces a floating-point result.

Example

```
nix-repl> 5 / 3
1

nix-repl> 5.0 / 3
1.66667
```

2.7. String operations

2.7.1. String comparison

Nix provides the usual lexicographic comparison operations.

Example

```
nix-repl> "apple" == "banana"      # equality
false

nix-repl> "apple" != "banana"      # inequality
true

nix-repl> "apple" < "banana"       # comes alphabetically before?
true

nix-repl> "apple" <= "banana"      # equal or comes alphabetically before?
true

nix-repl> "apple" > "banana"       # comes alphabetically after?
false

nix-repl> "apple" >= "banana"      # equal or comes alphabetically after?
false
```

2.7.2. String concatenation

String concatenation uses the `+` operator.

Example

```
nix-repl> "Hello, " + "world!"
"Hello, world!"
```

2.7.3. String interpolation

You can use the `${variable}` syntax to insert the value of a variable within a string.

Example

```
nix-repl> name = "Wombat"

nix-repl> "Hi, I'm ${name}."
"Hi, I'm Wombat."
```

You cannot mix numbers and strings. Earlier we set `a = 7`, so the following expression fails.

Example

```
nix-repl> "My favourite number is ${a}."
error:
       … while evaluating a path segment

         at «string»:1:25:

            1| "My favourite number is ${a}."
             |                         ^

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

Nix does provide functions for converting between types; we'll see these in the next section.

2.7.4. Useful built-in functions for strings

Nix provides some built-in functions for working with strings; a few examples are shown below. For more information on these and other built-in functions, see the Nix Manual (https://nixos.org/manual/nix/stable/language/builtins).

How long is this string?

Example

```
nix-repl> builtins.stringLength "supercalifragilisticexpialidocious"
34
```

Given a starting position and a length, extract a substring. The first character of a string has index `0`.

Example

```
nix-repl> builtins.substring 3 6 "hayneedlestack"
"needle"
```

Convert an expression to a string.

Example

```
nix-repl> builtins.toString 7
"7"

nix-repl> builtins.toString (3*4 + 1)
"13"
```

2.8. Boolean operations

The usual boolean operators are available. Recall that earlier we set `a = 7` and `b = 3`.

Example

```
nix-repl> a == 7                    # equality test
true

nix-repl> b != 3                    # inequality
false

nix-repl> a > 12                    # greater than
false

nix-repl> b >= 2                    # greater than or equal
true

nix-repl> a < b                     # less than
false

nix-repl> b <= a                    # less than or equal
true

nix-repl> !true                     # logical negation
false

nix-repl> (3 * a == 21) && (a > b)   # logical AND
true

nix-repl> (b > a) || (b > 10)        # logical OR
false
```

One operator that might be unfamiliar to you is *logical implication*, which uses the symbol `→`. The expression `u → v` is equivalent to `!u || v`.

Example

```
nix-repl> u = false

nix-repl> v = true
```

```
nix-repl> u -> v
true

nix-repl> v -> u
false
```

2.9. Path operations

Any expression that contains a forward slash (/) *not* followed by a space is interpreted as a path. To refer to a file or directory relative to the current directory, prefix it with `./`. You can specify the current directory as `./.`

Example

```
nix-repl> ./file.txt
/home/amy/codeberg/nix-book/file.txt

nix-repl> ./.
/home/amy/codeberg/nix-book
```

2.9.1. Concatenating paths

Paths can be concatenated to produce a new path.

Example

```
nix-repl> /home/wombat + /bin/sh
/home/wombat/bin/sh

nix-repl> :t /home/wombat + /bin/sh
a path
```

Relative paths are made absolute when they are parsed, which occurs before concatenation. This is why the result in the example below is not `/home/wombat/file.txt`.

Example

```
nix-repl> /home/wombat + ./file.txt
/home/wombat/home/amy/codeberg/nix-book/file.txt
```

2.9.2. Concatenating a path + a string

A path can be concatenated with a string to produce a new path.

Example

```
nix-repl> /home/wombat + "/file.txt"
/home/wombat/file.txt

nix-repl> :t /home/wombat + "/file.txt"
a path
```

The Nix reference manual says that the string must not "have a string context" that refers to a store path. String contexts are beyond the scope of this book; for more information see https://nixos.org/manual/nix/stable/language/operators#path-concatenation.

2.9.3. Concatenating a string + a path

Strings can be concatenated with paths, but with a side-effect that may surprise you: if the path exists, the file is copied to the Nix store! The result is a string, not a path.

In the example below, you might expect the result to be `"home/wombat/file.nix"`. However, the file `file.txt` is copied to `/nix/store/gp8ba25gpwvbqizqfr58jr014gmv1hd8-file.txt` before concatenating it to the string.

Example

```
nix-repl> "/home/wombat" + ./file.txt
"/home/wombat/nix/store/gp8ba25gpwvbqizqfr58jr014gmv1hd8-file.txt"
```

When concatenating a string with a path, the path must exist.

Example

```
nix-repl> "/home/wombat" + ./no-such-file.txt
error (ignored): error: end of string reached
error: getting status of '/home/amy/codeberg/nix-book/no-such-file.txt': No such file
or directory
```

2.9.4. Useful built-in functions for paths

Nix provides some built-in functions for working with paths; a few examples are shown below. For more information on these and other built-in functions, see the Nix Manual (https://nixos.org/manual/nix/stable/language/builtins).

Does the path exist?

Example

```
nix-repl> builtins.pathExists ./index.html
```

```
true

nix-repl> builtins.pathExists /no/such/path
false
```

Get a list of the files in a directory, along with the type of each file.

Example

```
nix-repl> builtins.readDir ./.
{ ".envrc" = "regular"; ".git" = "directory"; ".gitignore" = "regular"; Makefile =
"regular"; images = "directory"; "index.html" = "regular"; "shell.nix" = "regular";
source = "directory"; themes = "directory"; "wombats-book-of-nix.pdf" = "regular"; }
```

Read the contents of a file into a string.

Example

```
nix-repl> builtins.readFile ./.envrc
"use nix\n"
```

2.10. List operations

2.10.1. List concatenation

Lists can be concatenated using the `++` operator.

Example

```
nix-repl> [ 1 2 3 ] ++ [ "apple" "banana" ]
[ 1 2 3 "apple" "banana" ]
```

2.10.2. Useful built-in functions for lists

Nix provides some built-in functions for working with lists; a few examples are shown below. For more information on these and other built-in functions, see the Nix Manual (https://nixos.org/manual/nix/stable/language/builtins).

Testing if an element appears in a list.

Example

```
nix-repl> fruit = [ "apple" "banana" "cantaloupe" ]

nix-repl> builtins.elem "apple" fruit
true

nix-repl> builtins.elem "broccoli" fruit
```

```
false
```

Selecting an item from a list by index. The first element in a list has index `0`.

Example

```
nix-repl> builtins.elemAt fruit 0
"apple"

nix-repl> builtins.elemAt fruit 2
"cantaloupe"
```

Determining the number of elements in a list.

Example

```
nix-repl> builtins.length fruit
3
```

Accessing the first element of a list.

Example

```
nix-repl> builtins.head fruit
"apple"
```

Dropping the first element of a list.

Example

```
nix-repl> builtins.tail fruit
[ "banana" "cantaloupe" ]
```

Functions are useful for working with lists. Functions will be introduced in Section 2.12, “Functions”, but the following examples should be somewhat self-explanatory.

Using a function to filter (select elements from) a list.

Example

```
nix-repl> numbers = [ 1 3 6 8 9 15 25 ]

nix-repl> isBig = n: n > 10                # is the number "big" (greater than 10)?

nix-repl> builtins.filter isBig numbers    # get just the "big" numbers
[ 15 25 ]
```

Applying a function to all the elements in a list.

Example

```
nix-repl> double = n: 2*n                # multiply by two

nix-repl> builtins.map double numbers    # double each element in the list
[ 2 6 12 16 18 30 50 ]
```

2.11. Attribute set operations

2.11.1. Selection

The `.` operator selects an attribute from a set.

Example

```
nix-repl> animal = { name = { first = "Professor"; last = "Paws"; }; age = 10; species
= "cat"; }

nix-repl> animal . age
10

nix-repl> animal . name . first
"Professor"
```

2.11.2. Query

We can use the `?` operator to find out if a set has a particular attribute.

Example

```
nix-repl> animal ? species
true

nix-repl> animal ? bicycle
false
```

2.11.3. Union

We can use the `//` operator to combine two attribute sets. Attributes in the right-hand set take preference.

Example

```
nix-repl> a = { x = 7; y = "hello"; }

nix-repl> b = { y = "wombat"; z = 3; }

nix-repl> a // b
```

```
{
  x = 7;
  y = "wombat";
  z = 3;
}
```

This is often used to "modify" one or more values in the set. Recall that Nix values are immutable, so the result is a new value (the original is not actually modified).

Example

```
nix-repl> animal // { species = "tiger"; }
{ age = 10; name = { ... }; species = "tiger"; }
```

2.11.4. Recursive attribute sets

An ordinary attribute set cannot refer to its own elements. To do this, you need a *recursive* attribute set.

Example

```
nix-repl> { x = 3; y = 4*x; }
error: undefined variable 'x'

       at «string»:1:16:

            1| { x = 3; y = 4*x; }
             |                ^

nix-repl> rec { x = 3; y = 4*x; }
{ x = 3; y = 12; }
```

2.11.5. Useful built-in functions for attribute sets

Nix provides some built-in functions for working with attribute sets; a few examples are shown below. For more information on these and other built-in functions, see the Nix Manual (https://nixos.org/manual/nix/stable/language/builtins).

Get an alphabetical list of the keys.

Example

```
nix-repl> builtins.attrNames animal
[ "age" "name" "species" ]
```

Get the values associated with each key, in alphabetical order by the key names.

Example

```
nix-repl> builtins.attrValues animal
[ 10 "Professor Paws" "cat" ]
```

What value is associated with a key?

Example

```
nix-repl> builtins.getAttr "age" animal
10
```

Does the set have a value for a key?

Example

```
nix-repl> builtins.hasAttr "name" animal
true

nix-repl> builtins.hasAttr "car" animal
false
```

Remove one or more keys and associated values from a set.

Example

```
nix-repl> builtins.removeAttrs animal [ "age" "species" ]
{ name = "Professor Paws"; }
```

Display an attribute set, including nested sets.

Example

```
nix-repl> builtins.toJSON animal
"{\"age\":10,\"name\":{\"first\":\"Professor\",\"last\":\"Paws\"},\"species\":\"cat\"}
"
```

2.12. Functions

2.12.1. Anonymous functions

Functions are defined using the syntax `parameter: expression`, where the *expression* typically involves the *parameter*. Consider the following example.

Example

```
nix-repl> x: x + 1
```

```
«lambda @ «string»:1:1»
```

We created a function that adds 1 to its input. However, it doesn't have a name, so we can't use it directly. Anonymous functions do have their uses, as we shall see shortly.

Note that the message printed by the Nix REPL when we created the function uses the term *lambda*. This derives from a branch of mathematics called *lambda calculus*. Lambda calculus was the inspiration for most functional languages such as Nix. Functional programmers often call anonymous functions "lambdas".

The Nix REPL confirms that the expression `x: x + 1` defines a function.

Example

```
nix-repl> :t x: x + 1
a function
```

2.12.2. Named functions and function application

How can we use a function? Recall from Section 2.2.8, "Functions" that functions can be treated like any other data type. In particular, we can assign it to a variable.

Example

```
nix-repl> f = x: x + 1

nix-repl> f
«lambda @ «string»:1:2»
```

Procedural languages such as C or Java often use parenthesis to apply a function to a value, e.g. `f(5)`. Nix, like lambda calculus and most functional languages, does not require parenthesis for function application. This reduces visual clutter when chaining a series of functions.

Now that our function has a name, we can use it.

Example

```
nix-repl> f 5
6
```

2.12.3. Multiple parameters using nested functions

Functions in Nix always have a single parameter. To define a calculation that requires more than one parameter, we create functions that return functions!

Example

```
nix-repl> add = a: (b: a+b)
```

We have created a function called `add`. When applied to a parameter `a`, it returns a new function that adds `a` to its input. Note that the expression `(b: a+b)` is an anonymous function. We never call it directly, so it doesn't need a name. Anonymous functions are useful after all!

I used parentheses to emphasise the inner function, but they aren't necessary. More commonly we would write the following.

Example

```
nix-repl> add = a: b: a+b
```

If we only supply one parameter to `add`, the result is a new function rather than a simple value. Invoking a function without supplying all of the expected parameters is called *partial application*.

Example

```
nix-repl> add 3             # Returns a function that adds 3 to any input
«lambda @ «string»:1:6»
```

Now let's apply `add 3` to the value `5`.

Example

```
nix-repl> (add 3) 5
8
```

In fact, the parentheses aren't needed.

Example

```
nix-repl> add 3 5
8
```

If you've never used a functional programming language, this all probably seems very strange. Imagine that you want to add two numbers, but you have a very unusual calculator labeled "add". This calculator never displays a result, it only produces more calculators! If you enter the value `3` into the "add" calculator, it gives you a second calculator labeled "add 3". You then enter `5` into the "add 3" calculator, which displays the result of the addition, `8`.

With that image in mind, let's walk through the steps again in the REPL, but this time in more detail. The function `add` takes a single parameter `a`, and returns a new function that takes a single parameter `b`, and returns the value `a + b`. Let's apply `add` to the value `3`, and give the resulting new function a name, `g`.

Example

```
nix-repl> g = add 3
```

The function `g` takes a single parameter and adds `3` to it. The Nix REPL confirms that `g` is indeed a function.

Example

```
nix-repl> :t g
a function
```

Now we can apply `g` to a number to get a new number.

Example

```
nix-repl> g 5
8
```

2.12.4. Multiple parameters using attribute sets

I said earlier that a function in Nix always has a single parameter. However, that parameter need not be a simple value; it could be a list or an attribute set. This approach is widely used in Nix, and the language has some special features to support it. This is an important topic, so we will cover it separately in Section 2.13, “Argument sets”.

2.13. Argument sets

An attribute set that is used as a function parameter is often called an *argument set*.

2.13.1. Set patterns

To specify an attribute set as a function parameter, we use a *set pattern*, which has the form

```
{ name1, name2, ... }
```

Note that while the key-value associations in attribute sets are separated by semi-colons, the key names in the attribute set _pattern are separated by commas. Here’s an example of a function that has an attribute set as an input parameter.

Example

```
nix-repl> greet = { first, last }: "Hello ${first} ${last}! May I call you ${first}?"

nix-repl> greet { first="Amy"; last="de Buitléir"; }
"Hello Amy de Buitléir! May I call you Amy?"
```

2.13.2. Optional parameters

We can make some values in an argument set optional by providing default values, using the syntax

`name ? value`. This is illustrated below.

Example

```
nix-repl> greet = { first, last ? "whatever-your-lastname-is", topic ? "Nix" }: "Hello
${first} ${last}! May I call you ${first}? Are you enjoying learning ${topic}?"

nix-repl> greet { first="Amy"; }
"Hello Amy whatever-your-lastname-is! May I call you Amy? Are you enjoying learning
Nix?"

nix-repl> greet { first="Amy"; topic="Mathematics";}
"Hello Amy whatever-your-lastname-is! May I call you Amy? Are you enjoying learning
Mathematics?"
```

2.13.3. Variadic attributes

A function can allow the caller to supply argument sets that contain "extra" values. This is done with the special parameter `...`.

Example

```
nix-repl> formatName = { first, last, ... }: "${first} ${last}"
```

One reason for doing this is to allow the caller to pass the same argument set to multiple functions, even though each function may not need all of the values.

Example

```
nix-repl> person = { first="Joe"; last="Bloggs"; address="123 Main Street"; }

nix-repl> formatName person
"Joe Bloggs"
```

Another reason for allowing variadic arguments is when a function calls another function, supplying the same argument set. An example is shown in Section 2.13.4, “@-patterns”.

2.13.4. @-patterns

It can be convenient for a function to be able to reference the argument set as a whole. This is done using an *@-pattern*.

Example

```
nix-repl> formatPoint = p@{ x, y, ... }: builtins.toXML p

nix-repl> formatPoint { x=5; y=3; z=2; }
"<?xml version='1.0' encoding='utf-8'?>\n<expr>\n  <attrs>\n    <attr name=\"x\">\n
<int value=\"5\" />\n    </attr>\n    <attr name=\"y\">\n      <int value=\"3\" />\n
```

```
</attr>\n    <attr name=\"z\">\n      <int value=\"2\" />\n    </attr>\n
</attrs>\n</expr>\n"
```

Alternatively, the @-pattern can appear *after* the argument set, as in the example below.

Example

```
nix-repl> formatPoint = { x, y, ... } @ p: builtins.toXML p
```

An @-pattern is the only way a function can access variadic attributes, so they are often used together. In the example below, the function `greet` passes its argument set, including the variadic arguments, to the function `confirmAddress`.

Example

```
nix-repl> confirmAddress = { address, ... }: "Do you still live at ${address}?"

nix-repl> greet = args@{ first, last, ... }: "Hello ${first}. " + confirmAddress args

nix-repl> greet person
"Hello Joe. Do you still live at 123 Main Street?"
```

2.14. Derivations

Derivations can be created using the primitive built-in `derivation` function. It takes the following arguments.

- `system` (e.g. `x86_64-linux`).
- `name`, the package name.
- `builder`, the executable that builds the package. Attributes are passed to the builder as environment variables.
- `args` (optional), command line arguments to be passed to the builder.
- `outputs` (optional, defaults to `out`), output names. Nix will pass them to the builder as environment variables containing the output paths.

  ```
  d = derivation { name = "myname"; builder = "mybuilder"; system = "mysystem"; }
  ```

Example

```
nix-repl> d = derivation { name = "myname"; builder = "mybuilder"; system =
"mysystem"; }
```

In place of using `derivation`, it is generally more convenient to use `stdenv.mkDerivation`, which will be be introduced in Section 3.5, “`stdenv.mkDerivation`”

2.14.1. Instantiation vs Realisation

When a derivation is *instantiated* (*evaluated*), the Nix expression is parsed and interpreted. The result is a `.drv` file containing a derivation set. The package is not built in this step.

The package itself is created when the derivation is *built* (*realised*). Any dependencies are also built at this time.

2.14.2. Instantiate (evaluate) a derivation

Using the derivation "d" created above...

Example

```
nix-repl> d
«derivation /nix/store/z3hhlxbckx4g3n9sw91nnvlkjvyw754p-myname.drv»
```

That `.drv` file is plain text; it contains the derivation in a different format. The mysterious sequence of characters in the filename is a hash of the derivation. The hash is unique to this derivation. We could have multiple derivations for a package, perhaps different versions or with different options enabled, but each one would have a unique hash. Any changes to the derivation would result in a new hash. Using the hash, Nix can tell the different derivations apart, and avoid rebuilding a derivation that has already been built.

We can inspect the derivation in the Nix repl.

Example

```
nix-repl> d.drvAttrs
{ builder = "mybuilder"; name = "myname"; system = "mysystem"; }
```

We can examine the `.drv` file.

Example

```
$ cat /nix/store/z3hhlxbckx4g3n9sw91nnvlkjvyw754p-myname.drv
Derive([("out","/nix/store/40s0qmrfb45vlh6610rk29ym318dswdr-
myname","","")],[],[],"mysystem","mybuilder",[],[("builder","mybuilder"),("name","myna
me"),("out","/nix/store/40s0qmrfb45vlh6610rk29ym318dswdr-
myname"),("system","mysystem")])%
```

Or get a nicely formatted version.

Example

```
$ nix show-derivation /nix/store/z3hhlxbckx4g3n9sw91nnvlkjvyw754p-myname.drv
{
  "/nix/store/z3hhlxbckx4g3n9sw91nnvlkjvyw754p-myname.drv": {
    "outputs": {
```

```
      "out": {
        "path": "/nix/store/40s0qmrfb45vlh6610rk29ym318dswdr-myname"
      }
    },
    "inputSrcs": [],
    "inputDrvs": {},
    "system": "mysystem",
    "builder": "mybuilder",
    "args": [],
    "env": {
      "builder": "mybuilder",
      "name": "myname",
      "out": "/nix/store/40s0qmrfb45vlh6610rk29ym318dswdr-myname",
      "system": "mysystem"
    }
  }
}
```

2.14.3. Find out where the package would be (or was) installed

Using the derivation "d" created above...

Example

```
nix-repl> d.outPath
"/nix/store/40s0qmrfb45vlh6610rk29ym318dswdr-myname"

nix-repl> builtins.toString d     # also works
"/nix/store/40s0qmrfb45vlh6610rk29ym318dswdr-myname"
```

2.14.4. Build (realise) a derivation.

In the Nix REPL, using the derivation "d" created above...

Example

```
nix-repl> :b d
error: a 'mysystem' with features {} is required to build
'/nix/store/z3hhlxbckx4g3n9sw91nnvlkjvyw754p-myname.drv', but I am a 'x86_64-linux'
with features {benchmark, big-parallel, kvm, nixos-test}
```

Of course, this example failed to build because the builder and system are fake. We can achieve the same result at the command line.

Example

```
nix-build /nix/store/z3hhlxbckx4g3n9sw91nnvlkjvyw754p-myname.drv
this derivation will be built:
  /nix/store/z3hhlxbckx4g3n9sw91nnvlkjvyw754p-myname.drv
```

```
error: a 'mysystem' with features {} is required to build
'/nix/store/z3hhlxbckx4g3n9sw91nnvlkjvyw754p-myname.drv', but I am a 'x86_64-linux'
with features {benchmark, big-parallel, kvm, nixos-test}
```

2.14.5. Remove a derivation

Delete a path from the store **if it is safe to do so** (i.e. if it is eligible for garbage collection).

Example

```
nix-store --delete /nix/store/rc3n0lc4aijyi9wwpmcss7hbxryz6bnh-foo
```

2.15. If expressions

The conditional construct in Nix is an *expression*, not a *statement*. Since expressions must have values in all cases, you must specify both the `then` and the `else` branch.

Example

```
nix-repl> a = 7

nix-repl> b = 3

nix-repl> if a > b then "yes" else "no"
"yes"
```

2.16. Let expressions

A `let` expression defines a value with a local scope.

Example

```
nix-repl> let x = 3; in x*x
9

nix-repl> let x = 3; y = 2; in x*x + y
11
```

You can also nest `let` expressions. The previous expression is equivalent to the following.

Example

```
nix-repl> let x = 3; in let y = 2; in x*x + y
11
```

A variable defined inside a `let` expression will "shadow" an outer variable with the

same name.

Example

```
nix-repl> x = 100

nix-repl> let x = 3; in x*x
9

nix-repl> let x = 3; in let x = 7; in x+1
8
```

A variable in a let expression can refer to another variable in the expression. This is similar to how recursive attribute sets work.

Example

```
nix-repl> let x = 3; y = x + 1; in x*y
12
```

2.17. With expressions

A `with` expression is somewhat similar to a `let` expression, but it brings all of the associations in an attribute set into scope.

Example

```
nix-repl> point = { x1 = 3; x2 = 2; }

nix-repl> with point; x1*x1 + x2
11
```

Unlike a `let` expression, a variable defined inside a `with` expression will *not* "shadow" an outer variable with the same name.

Example

```
nix-repl> name = "Amy"

nix-repl> animal = { name = "Professor Paws"; age = 10; species =
"cat"; }

nix-repl> with animal; "Hello, " + name
"Hello, Amy"
```

However, you can refer to the variable in the inner scope using the attribute selection operator (`.`).

Example

```
nix-repl> with animal; "Hello, " + animal.name
"Hello, Professor Paws"
```

2.18. Inherit

The `inherit` keyword causes the specified attributes to be bound to whatever variables with the same name happen to be in scope.

When defining a set or in a let-expression it is often convenient to copy variables from the surrounding lexical scope (e.g., when you want to propagate attributes). This can be shortened using `inherit`.

For instance,

```
let x = 123; in
{
  inherit x;
  y = 456;
}
```

is equivalent to

```
let x = 123; in
{
  x = x;
  y = 456;
}
```

2.19. Import

The built-in `import` function provides a way to parse a `.nix` file.

a.nix

```
{
  message = "You successfully imported me!";
  b = 12;
}
```

Example

```
nix-repl> a = import ./a.nix

nix-repl> a.message
```

```
"You successfully imported me!"

nix-repl> a.b
12
```

If path supplied to the `import` function is a directory, the file default.nix in that directory is imported.

The scope of the imported file does not inherit the scope of the importer.

b.nix

```
x + 7
```

Example

```
nix-repl> x = 12

nix-repl> y = import ./b.nix

nix-repl> y
error:
       … while calling the 'import' builtin
         at «string»:1:2:
            1|  import ./b.nix
             |  ^

       error: undefined variable 'x'
       at /home/amy/codeberg/nix-book/b.nix:1:1:
            1| x + 7
             | ^
            2|
```

So to pass information when importing something, use a function.

c.nix

```
x: x + 7
```

Example

```
nix-repl> f = import ./c.nix

nix-repl> f 12
19
```

The imported file may import other files, which may in turn import more files.

Chapter 3. Nixpkgs

The Nix Packages collection (nixpkgs) is a large set of Nix expressions containing hundreds of software packages. The collection includes functions, definitions and other tools provided by Nix for creating, using, and maintaining Nix packages. This chapter will explore some of the most useful tools provided by nixpkgs.

In order to use nixpkgs, you must import it. As discussed in Section 2.2.5, “Paths”, enclosing a path in angle brackets, e.g. <nixpkgs> causes the directories listed in the environment variable NIX_PATH to be searched for the given file or directory name. In the REPL, the command `:l <nixpkgs>` will import nixpkgs.

Example

```
nix-repl> :l <nixpkgs>
Added 25694 variables.
```

Alternatively, you can automatically import nixpkgs when you enter the REPL using the command `nix repl '<nixpkgs>'`.

Within a Nix flake or module, you would use the `import` command. For example,

```
with (import <nixpkgs> {}); ...
```

When you import nixpkgs, you are importing a file, which imports other files, which import still more files. You can find the location of the nixpkgs directory.

```
nix-repl> <nixpkgs>
/nix/store/5izw1shpjxb9qhlf67bx427cih5czj8w-source
```

In that directory is a file called `default.nix`, which is what is actually imported. That file contains an import directive, which triggers more imports, and so on. As a result, tens of thousands of variables are added to the scope. Don’t worry; this doesn’t cause those hundreds of packages to be installed on your system. It merely gives you access to the recipes for those packages, plus some tools for working with packages.

This chapter will focus on a few especially useful Nixpkgs library functions. You can find a full list in the Nixpkgs manual.

3.1. `lib.genAttrs`

The function `lib.genAttrs` generates an attribute set by mapping a function over a list of attribute names. It is an alias for `lib.attrsets.genAttrs`.

It takes two arguments: - names of values in the resulting attribute set - a function which, given the

name of the attribute, returns the attribute's value

Example

```
nix-repl> lib.genAttrs [ "x86_64-linux" "aarch64-linux" ] (system: "some definitions
for ${system}")
{
  aarch64-linux = "some definitions for aarch64-linux";
  x86_64-linux = "some definitions for x86_64-linux";
}
```

As we shall see later, this is very useful when writing flakes.

3.2. `lib.getExe` and `lib.getExe'`

The function `lib.getExe` returns the path to the main program of a package. It is an alias for `lib.meta.getExe`.

Example

```
nix-repl> system = "x86_64-linux"

nix-repl> pkgs = import <nixpkgs> { inherit system; }

nix-repl> lib.getExe pkgs.hello
"/nix/store/s9p0adfpzarzfa5kcnqhwargfwiq8qmj-hello-2.12.2/bin/hello"
```

The function `lib.getExe'` returns the path to the specified program of a package. It is an alias for `lib.meta.getExe'`.

Example

```
nix-repl> lib.getExe' pkgs.imagemagick "convert"
"/nix/store/rn6ck85zkpkgdnm00jmn762z22wz86w6-imagemagick-7.1.2-3/bin/convert"
```

3.3. `lib.systems.flakeExposed`

This attribute is a list of systems that can support flakes.

Example

```
nix-repl> lib.systems.flakeExposed
[
  "x86_64-linux"
  "aarch64-linux"
  "x86_64-darwin"
  "armv6l-linux"
  "armv7l-linux"
```

```
  "i686-linux"
  "aarch64-darwin"
  "powerpc64le-linux"
  "riscv64-linux"
  "x86_64-freebsd"
]
```

This attribute is considered experimental and is subject to change.

3.4. `pkgs.mkShell` and `pkgs.mkShellNoCC`

The function `pkgs.mkShell` defines a Bash environment. It is a wrapper around `stdenv.mkDerivation`, discussed in Section 3.5, “`stdenv.mkDerivation`”.

The following attributes are accepted, along with all attributes of `stdenv.mkDerivation`.

attribute	description	default
`name`	the name of the derivation	`nix-shell`
`packages`	executable packages to add the shell	`[]`
`inputsFrom`	build dependencies to add to the shell	`[]`
`shellHook`	Bash statements to be executed by the shell	`""`

If you don’t need a C compiler, you can use `mkShellNoCC` instead.

3.5. `stdenv.mkDerivation`

The function `pkgs.mkShell` is a wrapper around the primitive `derivation` function, discussed in Section 2.14, “Derivations”. Some of the commonly used attributes are listed below.

attribute	description
`pname`	package name
`version`	version number. Use semantic versioning, i.e., MAJOR.MINOR.PATCH
`src`	location of the source files
`unpackPhase`	Bash command to copy/unpack the source files. If set to `"true"`, copies/unpacks all files in `src`, including subdirectories.
`buildPhase`	Bash commands to build the package. If no action is required, use the no-op `":"` command.
`installPhase`	Bash commands to install the package into the Nix store.

Older Nix flakes combined the package name and version number into a single `name` attribute; however, this is now discouraged.

A complete list of phases is available in the Nixpkgs manual. Additional arguments are also listed in

the Nixpkgs manual.

Chapter 4. Hello, flake!

Before learning to write Nix flakes, let's learn how to use them. I've created a simple example of a flake in this git repository: https://codeberg.org/mhwombat/hello-flake. To run this flake, you don't need to install anything; simply run the command below. The first time you use a flake, Nix has to fetch and build it, which may take time. Subsequent invocations should be instantaneous.

```
$ nix run "git+https://codeberg.org/mhwombat/hello-flake"
Hello from your flake!
```

That's a lot to type every time we want to use this package. Instead, we can enter a shell with the package available to us, using the `nix shell` command.

```
$ nix shell "git+https://codeberg.org/mhwombat/hello-flake"
```

In this shell, the command is on our `$PATH`, so we can execute the command by name.

```
$ hello-flake
Hello from your flake!
```

Nix didn't *install* the package; it merely built and placed it in a directory called the "Nix store". Thus we can have multiple versions of a package without worrying about conflicts. We can find out the location of the executable, if we're curious.

```
$ which hello-flake
/nix/store/9v2qpdkzfwqh8sqc0l6zglb4n0ah58aj-hello-flake/bin/hello-flake
```

Once we exit that shell, the `hello-flake` command is no longer directly available.

```
$ exit
$ hello-flake     # Fails outside development shell
bash: line 24: hello-flake: command not found
```

However, we can still run the command using the store path we found earlier. That's not particularly convenient, but it does demonstrate that the package remains in the store for future use.

```
$ /nix/store/9v2qpdkzfwqh8sqc0l6zglb4n0ah58aj-hello-flake/bin/hello-flake
Hello from your flake!
```

4.1. Flake outputs

You can find out what packages and apps a flake provides using the `nix flake show` command.

```
$ nix flake show --all-systems git+https://codeberg.org/mhwombat/hello-flake
git+https://codeberg.org/mhwombat/hello-
flake?ref=refs/heads/main&rev=67cb12b62400d9376148b52e78f64b56ef46e60b
├─────apps
│    ├─────aarch64-darwin
│    │    ├─────default: app: no description
│    │    └─────hello: app: no description
│    ├─────aarch64-linux
│    │    ├─────default: app: no description
│    │    └─────hello: app: no description
│    ├─────armv6l-linux
│    │    ├─────default: app: no description
│    │    └─────hello: app: no description
│    ├─────armv7l-linux
│    │    ├─────default: app: no description
│    │    └─────hello: app: no description
│    ├─────i686-linux
│    │    ├─────default: app: no description
│    │    └─────hello: app: no description
│    ├─────powerpc64le-linux
│    │    ├─────default: app: no description
│    │    └─────hello: app: no description
│    ├─────riscv64-linux
│    │    ├─────default: app: no description
│    │    └─────hello: app: no description
│    ├─────x86_64-darwin
│    │    ├─────default: app: no description
│    │    └─────hello: app: no description
│    ├─────x86_64-freebsd
│    │    ├─────default: app: no description
│    │    └─────hello: app: no description
│    └─────x86_64-linux
│         ├─────default: app: no description
│         └─────hello: app: no description
└─────packages
     ├─────aarch64-darwin
     │    ├─────default: package 'hello-flake'
     │    └─────hello: package 'hello-flake'
     ├─────aarch64-linux
     │    ├─────default: package 'hello-flake'
     │    └─────hello: package 'hello-flake'
     ├─────armv6l-linux
     │    ├─────default: package 'hello-flake'
     │    └─────hello: package 'hello-flake'
     ├─────armv7l-linux
     │    ├─────default: package 'hello-flake'
```

```
│   └─────hello: package 'hello-flake'
├─────i686-linux
│   ├─────default: package 'hello-flake'
│   └─────hello: package 'hello-flake'
├─────powerpc64le-linux
│   ├─────default: package 'hello-flake'
│   └─────hello: package 'hello-flake'
├─────riscv64-linux
│   ├─────default: package 'hello-flake'
│   └─────hello: package 'hello-flake'
├─────x86_64-darwin
│   ├─────default: package 'hello-flake'
│   └─────hello: package 'hello-flake'
├─────x86_64-freebsd
│   ├─────default: package 'hello-flake'
│   └─────hello: package 'hello-flake'
└─────x86_64-linux
    ├─────default: package 'hello-flake'
    └─────hello: package 'hello-flake'
```

Examining the output of this command, we see that this flake supports multiple architectures (aarch64-darwin, aarch64-linux, x86_64-darwin and x86_64-linux) and provides both a package and an app called `hello`.

Chapter 5. The hello-flake repo

Let's clone the repository and see how the flake is defined.

```
$ git clone https://codeberg.org/mhwombat/hello-flake
Cloning into 'hello-flake'...
$ cd hello-flake
$ ls
flake.lock
flake.nix
hello-flake
LICENSE
README.md
```

This is a simple repo with just a few files. Like most git repos, it includes `LICENSE`, which contains the software license, and `README.md` which provides information about the repo.

The `hello-flake` file is the executable we ran earlier. This particular executable is just a shell script, so we can view it. It's an extremely simple script with just two lines.

hello-flake

```
#!/usr/bin/env sh

echo "Hello from your flake!"
```

Now that we have a copy of the repo, we can execute this script directly.

```
$ ./hello-flake
Hello from your flake!
```

Not terribly exciting, I know. But starting with such a simple package makes it easier to focus on the flake system without getting bogged down in the details. We'll make this script a little more interesting later.

Let's look at another file. The file that defines how to package a flake is always called `flake.nix`.

flake.nix

```
{
  description = "a very simple and friendly flake";

  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs";
    flake-parts.url = "github:hercules-ci/flake-parts";
  };

```

```
 9   outputs = inputs@{ self, nixpkgs, flake-parts, ... }:
10     flake-parts.lib.mkFlake { inherit inputs; } {
11       systems = nixpkgs.lib.systems.flakeExposed;
12
13       perSystem = { self', pkgs, ... }: {
14         packages = rec {
15           hello = pkgs.stdenv.mkDerivation rec {
16             name = "hello-flake";
17
18             src = ./.;
19
20             unpackPhase = "true";
21
22             buildPhase = ":";
23
24             installPhase =
25               ''
26                 mkdir -p $out/bin
27                 cp $src/hello-flake $out/bin/hello-flake
28                 chmod +x $out/bin/hello-flake
29               '';
30           };
31           default = hello;
32         }; # packages
33
34         apps = rec {
35           hello = {
36             type = "app";
37             program = pkgs.lib.getExe' self'.packages.hello "hello-flake";
38           };
39
40           default = hello;
41         }; # apps
42       }; # perSystem
43     }; # mkFlake
44 }
```

If this is your first time seeing a flake definition, it probably looks intimidating. Flakes are written in the Nix language, introduced in Chapter 2, *The Nix language*. However, you don't really need to know Nix to follow this example. For now, I'd like to focus on the inputs section.

```
inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs";
    flake-utils.url = "github:numtide/flake-utils";
  };
```

There are just two entries, one for `nixpkgs` and one for `flake-utils`. The first one, `nixpkgs` refers to the collection of standard software packages that can be installed with the Nix package manager. The second, `flake-utils`, is a collection of utilities that simplify writing flakes. The important thing

to note is that the `hello-flake` package *depends* on `nixpkgs` and `flake-utils`.

Finally, let's look at `flake.lock`, or rather, just part of it.

flake.lock

```
{
  "nodes": {
    "flake-parts": {
      "inputs": {
        "nixpkgs-lib": "nixpkgs-lib"
      },
      "locked": {
        "lastModified": 1769996383,
        "narHash": "sha256-AnYjnFWgS49RlqX7LrC4uA+sCCDBj0Ry/WOJ5XWAsa0=",
        "owner": "hercules-ci",
        "repo": "flake-parts",
        "rev": "57928607ea566b5db3ad13af0e57e921e6b12381",
        "type": "github"
      },
      "original": {
        "owner": "hercules-ci",
        "repo": "flake-parts",
        "type": "github"
      }
    },
    "nixpkgs": {
      "locked": {
        "lastModified": 1770131990,
        "narHash": "sha256-Kn63KtkmiBeZ4K/U5aHzh+dtnEvfY764KRbLgblluME=",
        "owner": "NixOS",
        "repo": "nixpkgs",
        "rev": "c248f2ae64aa2b3c11afe3b50dd49c06885759c4",
        "type": "github"
      },
      "original": {
        "owner": "NixOS",
        "repo": "nixpkgs",
        "type": "github"
      }
    },
    "nixpkgs-lib": {
      "locked": {
        "lastModified": 1769909678,
        "narHash": "sha256-cBEymOf4/o3FD5AZnzC3J9hLbiZ+QDT/KDuyHXVJOpM=",
        "owner": "nix-community",
. . .
```

If `flake.nix` seemed intimidating, then this file looks like an invocation for Cthulhu. The good news is that this file is automatically generated; you never need to write it. It contains information about

all of the dependencies for the flake, including where they came from, the exact version/revision, and hash. This lockfile *uniquely* specifies all flake dependencies, (e.g., version number, branch, revision, hash), so that *anyone, anywhere, any time, can re-create the exact same environment that the original developer used.*

No more complaints of "but it works on my machine!". That is the benefit of using flakes.

Chapter 6. Flake structure

The basic structure of a flake is shown below.

```
{
  description = package description
  inputs = dependencies
  outputs = what the flake produces
  nixConfig = advanced configuration options
}
```

The `description` part is self-explanatory; it's just a string. You probably won't need `nixConfig` unless you're doing something fancy. I'm going to focus on what goes into the `inputs` and `outputs` sections, and highlight some of the things I found confusing when I began using flakes.

6.1. Inputs

This section specifies the dependencies of a flake. It's an *attribute set*; it maps keys to values.

To ensure that a build is reproducible, the build step runs in a *pure* environment with no network access. Therefore, any external dependencies must be specified in the "inputs" section so they can be fetched in advance (before we enter the pure environment).

Each entry in this section maps an input name to a *flake reference*. This commonly takes the following form.

```
NAME.url = URL-LIKE-EXPRESSION
```

As a first example of a flake reference, all (almost all?) flakes depend on "nixpkgs", which is a large Git repository of programs and libraries that are pre-packaged for Nix. We can write that as

```
nixpkgs.url = "github:NixOS/nixpkgs/nixos-version";
```

where *version* is replaced with the version number that you used to build the package, e.g. `22.11`. Information about the latest nixpkgs releases is available at https://status.nixos.org/. You can also write the entry without the version number

```
nixpkgs.url = "github:NixOS/nixpkgs/nixos";
```

or more simply,

```
nixpkgs.url = "nixpkgs";
```

You might be concerned that omitting the version number would make the build non-reproducible. If someone else builds the flake, could they end up with a different version of nixpkgs? No! remember that the lockfile (`flake.lock`) *uniquely* specifies all flake inputs.

Git and Mercurial repositories are the most common type of flake reference, as in the examples below.

A Git repository

```
git+https://github.com/NixOS/patchelf
```

A specific branch of a Git repository

```
git+https://github.com/NixOS/patchelf?ref=master
```

A specific revision of a Git repository

```
git+https://github.com/NixOS/patchelf?ref=master&rev=f34751b88bd07d7f44f5cd3200fb4122bf916c7e
```

A tarball

```
https://github.com/NixOS/patchelf/archive/master.tar.gz
```

You can find more examples of flake references in the Nix Reference Manual.

Although you probably won't need to use it, there is another syntax for flake references that you might encounter. This example

```
inputs.import-cargo = {
  type = "github";
  owner = "edolstra";
  repo = "import-cargo";
};
```

is equivalent to

```
inputs.import-cargo.url = "github:edolstra/import-cargo";
```

Each of the `inputs` is fetched, evaluated and passed to the `outputs` function as a set of attributes with the same name as the corresponding input.

6.2. Outputs

This section is a function that essentially returns the recipe for building the flake.

We said above that `inputs` are passed to the `outputs`, so we need to list them as parameters. This example references the `import-cargo` dependency defined in the previous example.

```
outputs = { self, nixpkgs, import-cargo }: {
  definitions for outputs
};
```

So what actually goes in the highlighted section? That depends on the programming languages your software is written in, the build system you use, and more. There are Nix functions and tools that can simplify much of this, and new, easier-to-use ones are released regularly. We'll look at some of these in the next section.

Chapter 7. A generic flake

The previous section presented a very high-level view of flakes, focusing on the basic structure. In this section, we will add a bit more detail.

Flakes are written in the Nix programming language, which is a functional language. As with most programming languages, there are many ways to achieve the same result. Below is an example you can follow when writing your own flakes. I'll explain the example in some detail.

```
{
  description = "brief package description";

  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs";
    ...other dependencies... ❶
  };

  outputs = { self, nixpkgs, ...other dependencies... ❷ }: {

    devShells = shell definitions; ❸

    packages = package definitions; ❹

    apps = app definitions; ❺

  };
}
```

We discussed how to specify flake inputs ❶ in the previous section, so this part of the flake should be familiar. Remember also that any dependencies in the input section should also be listed at the beginning of the outputs section ❷.

The `devShells` attribute ❸ specifies the environment that should be available when doing development on the package. This includes any tools (e.g., compilers and other language-specific build tools and packages). If you don't need a special development environment, you can omit this section.

The `packages` attribute ❹ defines the packages that this flake provides. The definition depends on the programming languages your software is written in, the build system you use, and more. There are Nix functions and tools that can simplify much of this, and new, easier-to-use ones are released regularly. These functions are very language-specific, and not always well-documented. We will see examples for some languages later in the tutorial. In general, I recommend that you do a web search for "nix *language-name*", and try to find resources that were written or updated recently.

The `apps` attribute ❺ identifies any applications provided by the flake. In particular, it identifies the default executable that `nix run` will run if you don't specify an app.

If we want the development shell, packages, and apps to be available for multiple systems (e.g.,

`x86_64-linux`, `aarch64-linux`, `x86_64-darwin`, and `aarch64-darwin`), we need to provide a definition for each of those systems. The result *could* be an outputs section like the one shown below.

```
outputs = { self, nixpkgs, ...other dependencies... ❷ }: {

  devShells.x86_64-linux.default = ...;
  devShells.aarch64-linux.default = ...;
  . . .

  packages.x86_64-linux.my-app = ...;
  packages.aarch64-linux.my-app = ...;
  . . .

  apps.x86_64-linux.my-app = ...;
  apps.aarch64-linux.my-app = ...;
  . . .

  apps.x86_64-linux.default = ...;
  apps.aarch64-linux.default = ...;
  . . .
};
```

You won't see definitions like that in most flakes. Typically the definitions for each shell, package or app would be identical, apart from a reference to the system name. There are techniques and tools that allow you to write a single definition and use it to automatically generate the definitions for multiple architectures. We will see some examples of this later in the tutorial.

Below is a list of some functions that are commonly used in the output section.

General-purpose

- `mkDerivation` is especially useful for the typical `./configure; make; make install` scenario. It's customisable.
- `mkShell` simplifies writing a development shell definition.
- `writeShellApplication` creates an executable shell script which also defines the appropriate environment.

Python

- `buildPythonApplication`
- `buildPythonPackage`.

Haskell

- `mkDerivation` (Haskell version, which is a wrapper around the standard environment version)
- `developPackage`
- `callCabal2Nix`.

Noogle allows you to search for documentation for Nix functions and other

> definitions.

Chapter 8. Another look at hello-flake

Now that we have a better understanding of the structure of `flake.nix`, let's have a look at the one we saw earlier, in the `hello-flake` repo. If you compare this flake definition to the colour-coded template presented in Chapter 7, *A generic flake*, most of it should look familiar.

flake.nix

```
{
  description = "a very simple and friendly flake";

  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs";
    flake-utils.url = "github:numtide/flake-utils";
  };

  outputs = { self, nixpkgs, flake-utils }:
    flake-utils.lib.eachDefaultSystem (system:
      let
        pkgs = import nixpkgs { inherit system; };
      in
      {
        packages = rec {
          hello =
            . . .
            _SOME UNFAMILIAR STUFF_
            . . .
          };
          default = hello;
        };

        apps = rec {
          hello = flake-utils.lib.mkApp { drv = self.packages.${system}.hello; };
          default = hello;
        };
      }
    );
}
```

This `flake.nix` doesn't have a `devShells` section, because development on the current version doesn't require anything beyond the "bare bones" linux commands. Later we will add a feature that requires additional development tools.

Now let's look at the section I labeled `SOME UNFAMILIAR STUFF` and see what it does.

```
        packages = rec {
          hello = pkgs.stdenv.mkDerivation rec { ❶
            name = "hello-flake";
```

```
      src = ./.; ❷

      unpackPhase = "true";

      buildPhase = ":";

      installPhase =
        ''
          mkdir -p $out/bin ❸
          cp $src/hello-flake $out/bin/hello-flake ❹
          chmod +x $out/bin/hello-flake ❺
        '';
    };
```

This flake uses `mkDerivation` ❶ which is a very useful general-purpose package builder provided by the Nix standard environment. It's especially useful for the typical `./configure; make; make install` scenario, but for this flake we don't even need that.

The `name` variable is the name of the flake, as it would appear in a package listing if we were to add it to Nixpkgs or another package collection. The `src` variable ❷ supplies the location of the source files, relative to `flake.nix`. When a flake is accessed for the first time, the repository contents are fetched in the form of a tarball. The `unpackPhase` variable indicates that we do want the tarball to be unpacked.

The `buildPhase` variable is a sequence of Linux commands to build the package. Typically, building a package requires compiling the source code. However, that's not required for a simple shell script. So `buildPhase` consists of a single command, `:`, which is a no-op or "do nothing" command.

The `installPhase` variable is a sequence of Linux commands that will do the actual installation. In this case, we create a directory ❸ for the installation, copy the `hello-flake` script there ❹, and make the script executable ❺. The environment variable `$src` refers to the source directory, which we specified earlier ❷.

Earlier we said that the build step runs in a pure environment to ensure that builds are reproducible. This means no Internet access; indeed no access to any files outside the build directory. During the build and install phases, the only commands available are those provided by the Nix standard environment and the external dependencies identified in the `inputs` section of the flake.

8.1. The Nix standard environment

I've mentioned the Nix standard environment before, but I didn't explain what it is. The standard environment, or `stdenv`, refers to the functionality that is available during the build and install phases of a Nix package (or flake). It includes the commands listed below.

- The GNU C Compiler, configured with C and C++ support.
- GNU coreutils (contains a few dozen standard Unix commands).
- GNU findutils (contains find).

- GNU diffutils (contains diff, cmp).
- GNU sed.
- GNU grep.
- GNU awk.
- GNU tar.
- gzip, bzip2 and xz.
- GNU Make.
- Bash.
- The patch command.
- On Linux, stdenv also includes the patchelf utility.

Only a few environment variables are available. The most interesting ones are listed below.

- `$name` is the package name.
- `$src` refers to the source directory.
- `$out` is the path to the location in the Nix store where the package will be added.
- `$system` is the system that the package is being built for.
- `$PWD` and `$TMP` both point to temporary build directories
- `$HOME` and `$PATH` point to nonexistent directories, so the build cannot rely on them.

For more information on the standard environment, see the Nixpkgs manual.

Chapter 9. A new flake from scratch

At last we are ready to create a flake from scratch! No matter what programming languages you normally use, I recommend that you start by reading the Section 9.1, "Bash" section. In it, I start with an extremely simple flake, and show how to improve and extend it.

The remaining sections in this chapter are very similar; read the one for your language of choice. If you're interested in a language that I haven't covered, feel free to suggest it by creating an issue.

9.1. Bash

In this section we will create a very simple flake, and then make several improvements. To follow along, open a new terminal shell. Start with an empty directory and create a git repository.

```
$ mkdir my-project
$ cd my-project
$ git init
Initialized empty Git repository in /home/amy/codeberg/nix-book/source/new-flake/bash-
flake/my-project/.git/
```

9.1.1. A simple Bash script

This will be a very simple development project, so that we can focus on how to use Nix. We want to package the following script.

cow-hello.sh

```
1 #!/usr/bin/env sh
2
3 cowsay "Hello from your flake!"
```

Add the script to your directory and make it executable. Let's test it.

```
$ chmod +x cow-hello.sh
$ ./cow-hello.sh     # Fails
./cow-hello.sh: line 3: cowsay: command not found
```

This probably failed on your machine — why? The `cow-hello.sh` script depends on `cowsay`, which isn't part of your default environment (unless you specifically configure Nix or NixOS to include it). We can create a suitable temporary environment for a quick test:

```
$ nix shell nixpkgs#cowsay
$ ./cow-hello.sh     # Succeeds
 ________________________
< Hello from your flake! >
```

```
 ------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

However, that approach becomes inconvenient once you have more dependencies. Let's exit to the original (default) environment, and see that `cowsay` is no longer available.

```
$ exit
$ ./cow-hello.sh     # Fails again
./cow-hello.sh: line 3: cowsay: command not found
```

For a single script like this, we could modify it by replacing the first line with a "Nix shebang", as shown later in Section 10.3, "Scripts"; then the script would run in any Nix environment. But for this exercise, we will package it as a "proper" development project.

9.1.2. Defining the development environment

We want to define the development environment we need for this project, so that we can recreate it at any time. Furthermore, anyone else who wants to work on our project will be able to recreate the same development environment we used. This avoids complaints of "but it works on my machine!"

We can use a function from Nixpkgs to create the environment, but we have to configure the package set for the system architecture. The function `nixpkgs` takes an attribute set as input, and returns a configured Nix package set that we can use. The only attribute we need to specify at this time is the system architecture name. For example, the following code creates a package set for the `x86_64-linux` architecture.

```
pkgs = import nixpkgs { system = "x86_64-linux"; };
```

In Section 3.4, "`pkgs.mkShell` and `pkgs.mkShellNoCC`" we learned that Nix provides the function called `mkShell`, which defines a Bash environment. We need to specify that the environment should provide `cowsay`.

```
pkgs.mkShell {
  packages = [ pkgs.cowsay ];
};
```

Now we're ready to write the flake. Create the file `flake.nix` as shown below. If you're not on an `x86_64-linux` system, modify the highlighted lines accordingly.

If you're not sure of the correct string so use for your system, run the following command.

```
nix eval --impure --raw --expr 'builtins.currentSystem'
```

flake.nix (version 1)

```
{
  description = "what does the cow say";

  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs";
  };

  outputs = { self, nixpkgs }: {

    devShells = {
      x86_64-linux.default =
        let
          pkgs = import nixpkgs { system = "x86_64-linux"; };
        in
          pkgs.mkShell {
            packages = [ pkgs.cowsay ];
          };
    }; # devShells

  }; # outputs
}
```

The `description` part is just a short description of the package. The `inputs` section should be familiar from Section 6.1, "Inputs". So far, the `outputs` section only defines a development environment (we'll add to it in Section 9.1.3, "Defining the package").

The repetition of `x86_64-linux` is undesirable. In Section 9.1.4, "Supporting multiple architectures" we will refactor the code to eliminate some duplication and make it more readable. For now, we will stick with the straightforward version.

Let's enter the shell.

```
$ nix develop # Fails
error: Path 'flake.nix' in the repository "/home/amy/codeberg/nix-book/source/new-
flake/bash-flake/my-project" is not tracked by Git.

       To make it visible to Nix, run:

       git -C "/home/amy/codeberg/nix-book/source/new-flake/bash-flake/my-project" add
"flake.nix"
```

Because we haven't added `flake.nix` to the git repository, it's essentially invisible to Nix. Let's correct that and try again. We'll also add the script to the git repository (otherwise we would get a

confusing message about the file being "missing").

```
$ git add flake.nix cow-hello.sh
$ nix develop
warning: Git tree '/home/amy/codeberg/nix-book/source/new-flake/bash-flake/my-project'
is dirty
warning: creating lock file "/home/amy/codeberg/nix-book/source/new-flake/bash-
flake/my-project/flake.lock":
• Added input 'nixpkgs':
    'github:NixOS/nixpkgs/c248f2ae64aa2b3c11afe3b50dd49c06885759c4?narHash=sha256-
Kn63KtkmiBeZ4K/U5aHzh%2BdtnEvfY764KRbLgbllluME%3D' (2026-02-03)
```

The warning is because the changes haven't been committed to the git repository yet. We changed `flake.nix` of course, but when we ran `nix develop` it automatically created a `flake.lock` file. Don't worry about the warnings for now. We can see that `cowsay` is now available, and our script runs.

```
$ ./cow-hello.sh     # Succeeds
 ________________________
< Hello from your flake! >
 ------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

9.1.3. Defining the package

We created an appropriate development environment, and tested our script. Now we are ready to package it.

In Section 3.5, "`stdenv.mkDerivation`" we learned that the function `pkgs.std.mkDerivation` provides a way to create a derivation by specifying the steps that need to be performed in each phase We specify the location of the source files. and use the standard unpack phase, which makes our source files available in `$src` during the build and installation. (We described the environment available during build and installation in Section 8.1, "The Nix standard environment".) We don't need to do anything in the build phase. In the install phase, we copy the script from `$src` to the output directory, `$out`, and make it executable. As with the development environment, we specify that `cowsay` is required.

```
pkgs.stdenv.mkDerivation {
  name = "cow-hello.sh";
  src = ./.;
  unpackPhase = "true";
  buildPhase = ":";
  installPhase =
    ''
```

```
      mkdir -p $out/bin
      cp $src/cow-hello.sh $out/bin
      chmod +x $out/bin/cow-hello.sh
    '';
  buildInputs = [ pkgs.cowsay ];
}; # mkDerivation
```

Here's the updated `flake.nix`. Again, change `x86_64-linux` if needed to match your system architecture.

flake.nix (version 2)

```
{
  description = "what does the cow say";

  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs";
  };

  outputs = { self, nixpkgs }: {

    devShells = {
      x86_64-linux.default =
        let
          pkgs = import nixpkgs { system = "x86_64-linux"; };
        in
          pkgs.mkShell {
            packages = [ pkgs.cowsay ];
          };
    }; # devShells

    packages = {
      x86_64-linux.default =
        let
          pkgs = import nixpkgs { system = "x86_64-linux"; };
        in
          pkgs.stdenv.mkDerivation {
            name = "cow-hello.sh";
            src = ./.;
            unpackPhase = "true";
            buildPhase = ":";
            installPhase =
              ''
                mkdir -p $out/bin
                cp $src/cow-hello.sh $out/bin
                chmod +x $out/bin/cow-hello.sh
              '';
            buildInputs = [ pkgs.cowsay ];
          }; # mkDerivation
    }; # packages
```

```
39
40   }; # outputs
41 }
```

Let's test the package. First, we should exit the development shell so we can verify that the flake dependencies are automatically loaded. Otherwise the script could use `cowsay` from the development shell. We'll also commit the changes to get rid of the warnings.

```
$ exit
$ git commit -am "define package"
[master (root-commit) e1437bd] define package
 3 files changed, 70 insertions(+)
 create mode 100755 cow-hello.sh
 create mode 100644 flake.lock
 create mode 100644 flake.nix
$ nix run # fails
this derivation will be built:
  /nix/store/fndlz4a874rxvvkfpzx2l9rrk1bij4yx-cow-hello.sh.drv
building '/nix/store/fndlz4a874rxvvkfpzx2l9rrk1bij4yx-cow-hello.sh.drv'...
/nix/store/xr0hcghnz1mbcqrn9raz1phnxxgkwdxf-cow-hello.sh/bin/cow-hello.sh: line 3:
cowsay: command not found
```

What went wrong? Although we made `cowsay` available in the runtime environment, the `cow-hello.sh` script can't find it.

For now, we can just edit the script during the build phase. We'll see another way to handle this in Section 10.5.1, "Access a top level package from the Nixpkgs/NixOS repo".

flake.nix (version 3)

```
 1 {
 2   description = "what does the cow say";
 3
 4   inputs = {
 5     nixpkgs.url = "github:NixOS/nixpkgs";
 6   };
 7
 8   outputs = { self, nixpkgs }: {
 9
10     devShells = {
11       x86_64-linux.default =
12         let
13           pkgs = import nixpkgs { system = "x86_64-linux"; };
14         in
15           pkgs.mkShell {
16             packages = [ pkgs.cowsay ];
17           };
18     }; # devShells
19
```

```
20      packages = {
21        x86_64-linux.default =
22          let
23            pkgs = import nixpkgs { system = "x86_64-linux"; };
24          in
25            pkgs.stdenv.mkDerivation {
26              name = "cow-hello.sh";
27              src = ./.;
28              unpackPhase = "true";
29              buildPhase = ":";
30              installPhase =
31                ''
32                  mkdir -p $out/bin
33                  cp $src/cow-hello.sh $out/bin
34                  chmod +x $out/bin/cow-hello.sh
35
36                  # modify the cow-hello.sh script so it can find cowsay
37                  COWSAY=$(type -p cowsay)
38                  sed "s_cowsay_"$COWSAY"_" --in-place $out/bin/cow-hello.sh
39                '';
40              buildInputs = [ pkgs.cowsay ];
41            }; # mkDerivation
42      }; # packages
43
44    }; # outputs
45  }
```

Let's try running the flake again.

```
$ git commit -am "fill in cowsay path"
[master a4ba36c] fill in cowsay path
 1 file changed, 4 insertions(+)
$ nix run
this derivation will be built:
  /nix/store/zamfq8lm55cc97kppahdkbqlzcbrrxa8-cow-hello.sh.drv
building '/nix/store/zamfq8lm55cc97kppahdkbqlzcbrrxa8-cow-hello.sh.drv'...
 ------------------------
< Hello from your flake! >
 ------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

9.1.4. Supporting multiple architectures

Congratulations! Our package is popular, and people want to run it on `aarch64-linux` systems. So now we need to add an entry to `packages`. Of course we want to test it on the new architecture, so

we'll add an entry to `devShells` as well.

flake.nix (version 4)

```
{
  description = "what does the cow say";

  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs";
  };

  outputs = { self, nixpkgs }: {

    devShells = {
      x86_64-linux.default =
        let
          pkgs = import nixpkgs { system = "x86_64-linux"; };
        in
          pkgs.mkShell {
            packages = [ pkgs.cowsay ];
          };

      aarch64-linux.default =
        let
          pkgs = import nixpkgs { system = "aarch64-linux"; };
        in
          pkgs.mkShell {
            packages = [ pkgs.cowsay ];
          };
    }; # devShells

    packages = {
      x86_64-linux.default =
        let
          pkgs = import nixpkgs { system = "x86_64-linux"; };
        in
          pkgs.stdenv.mkDerivation {
            name = "cow-hello.sh";
            src = ./.;
            unpackPhase = "true";
            buildPhase = ":";
            installPhase =
              ''
                mkdir -p $out/bin
                cp $src/cow-hello.sh $out/bin
                chmod +x $out/bin/cow-hello.sh

                # modify the cow-hello.sh script so it can find cowsay
                COWSAY=$(type -p cowsay)
                sed "s_cowsay_"$COWSAY"_" --in-place $out/bin/cow-hello.sh
              '';
```

```
48            buildInputs = [ pkgs.cowsay ];
49          }; # mkDerivation
50
51      aarch64-linux.default =
52        let
53          pkgs = import nixpkgs { system = "aarch64-linux"; };
54        in
55          pkgs.stdenv.mkDerivation {
56            name = "cow-hello.sh";
57            src = ./.;
58            unpackPhase = "true";
59            buildPhase = ":";
60            installPhase =
61              ''
62                mkdir -p $out/bin
63                cp $src/cow-hello.sh $out/bin
64                chmod +x $out/bin/cow-hello.sh
65
66                # modify the cow-hello.sh script so it can find cowsay
67                COWSAY=$(type -p cowsay)
68                sed "s_cowsay_"$COWSAY"_" --in-place $out/bin/cow-hello.sh
69              '';
70            buildInputs = [ pkgs.cowsay ];
71          }; # mkDerivation
72    }; # packages
73
74  }; # outputs
75 }
```

Let's make sure it still runs on our system.

```
$ git commit -am "suport aarch64-linux"
[master 716a142] suport aarch64-linux
 1 file changed, 30 insertions(+)
$ nix run
this derivation will be built:
  /nix/store/gs0k0km4v82m9aqsf6lhb3bxk1h0rara-cow-hello.sh.drv
building '/nix/store/gs0k0km4v82m9aqsf6lhb3bxk1h0rara-cow-hello.sh.drv'...

 ________________________
< Hello from your flake! >
 ------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Of course, we should also test on an `aarch64-linux` system. But the flake definition is rather long now. If we need to add even more architectures... ugh. Even worse, notice that the definitions for

`x86_64-linux` are almost identical to those for `aarch64-linux`. All that replication makes maintenance more difficult. What if we make a change to the definition for `aarch64-linux`, and forget to make the change to `x86_64-linux`?

It's time to refactor the code to eliminate duplication and make it more readable.

In Section 3.1, "`lib.genAttrs`", I introduced the `lib.genAttrs` function, and included a promising-looking example.

Example

```
nix-repl> lib.genAttrs [ "x86_64-linux" "aarch64-linux" ] (system: "some definitions
for ${system}")
{
  aarch64-linux = "some definitions for aarch64-linux";
  x86_64-linux = "some definitions for x86_64-linux";
}
```

This function generates an attribute set by mapping a function over a list of attribute names. It takes two arguments. The first argument is a list; the list elements will become names of values in the resulting attribute set. The second argument is a function that, given the name of the attribute, returns the attribute's value

In the example, we used a list of system architecture names as the first argument. For the second argument, we used a function that "pretended" to generate definitions.

What if we wrote a function which, given the name of the system architecture, would generate the development shell definition for us, and another function that would do the same for the package definition? Applying `lib.genAttrs` and the list of system architecture names to those functions would give us all the definitions we need for the outputs section.

The following function will generate a development shell definition. We will write the definition of `nixpkgsFor.${system}` shortly

```
system:
  let pkgs = nixpkgsFor.${system}; in {
    default = pkgs.mkShell {
        packages = [ pkgs.cowsay ];
    };
}
```

And this one will generate a package definition.

```
system:
  let pkgs = nixpkgsFor.${system}; in {
    default = pkgs.stdenv.mkDerivation {
      name = "cow-hello.sh";
      src = ./.;
```

```
      unpackPhase = "true";
      buildPhase = ":";
      installPhase =
        ''
          mkdir -p $out/bin
          cp $src/cow-hello.sh $out/bin
          chmod +x $out/bin/cow-hello.sh
        '';
      buildInputs = [ pkgs.cowsay ];
    }; # mkDerivation
}
```

So `lib.genAttrs [ "x86_64-linux" "aarch64-linux" ]` followed by the first function will give us the development shell definitions for both systems, and followed by the second function will give us the package definitions for both systems. We can make the flake more readable with the following definitions.

```
supportedSystems = [ "x86_64-linux" "aarch64-linux" ];
forEachSystem = nixpkgs.lib.genAttrs supportedSystems;
```

So `forEachSystem` is a function which takes one argument. That argument should be a function that, given the system name, generates the appropriate definition for that system, Now let's examine the definition of `nixpkgsFor.${system}`.

```
nixpkgsFor = forEachSystem (system: import nixpkgs { inherit system; });
```

Here we're using `forEachSystem` to access the appropriate nixpkgs for a system.

Putting everything together, we have a shiny new flake. You may want to compare it carefully to the original version, in order to reassure yourself that the definitions are equivalent.

flake.nix (version 5)

```
1  {
2    description = "what does the cow say";
3
4    inputs = {
5      nixpkgs.url = "github:NixOS/nixpkgs";
6    };
7
8    outputs = { self, nixpkgs }:
9      let
10       supportedSystems = [ "x86_64-linux" "aarch64-linux" ];
11       forEachSystem = nixpkgs.lib.genAttrs supportedSystems;
12       nixpkgsFor = forEachSystem (system: import nixpkgs { inherit system; });
13
14     in {
15
```

```
16      devShells = forEachSystem (system:
17        let pkgs = nixpkgsFor.${system}; in {
18          default = pkgs.mkShell {
19              packages = [ pkgs.cowsay ];
20          };
21      });
22
23      packages = forEachSystem (system:
24        let pkgs = nixpkgsFor.${system}; in {
25          default = pkgs.stdenv.mkDerivation {
26            name = "cow-hello.sh";
27            src = ./.;
28            unpackPhase = "true";
29            buildPhase = ":";
30            installPhase =
31              ''
32                mkdir -p $out/bin
33                cp $src/cow-hello.sh $out/bin
34                chmod +x $out/bin/cow-hello.sh
35
36                # modify the cow-hello.sh script so it can find cowsay
37                COWSAY=$(type -p cowsay)
38                sed "s_cowsay_"$COWSAY"_" --in-place $out/bin/cow-hello.sh
39              '';
40            buildInputs = [ pkgs.cowsay ];
41          }; # mkDerivation
42      }); # packages
43
44  }; # outputs
45 }
```

Note that if we need to support another architecture, we only need to add it to line 10. Let's verify that it runs on our system.

```
$ git commit -am "refactored the flake"
[master e3f53a1] refactored the flake
 1 file changed, 17 insertions(+), 47 deletions(-)
$ nix run
this derivation will be built:
  /nix/store/9hpnjl06rwdrr3bs84a1msfnn9jw2rx0-cow-hello.sh.drv
building '/nix/store/9hpnjl06rwdrr3bs84a1msfnn9jw2rx0-cow-hello.sh.drv'...
 ________________________
< Hello from your flake! >
 ------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

9.1.5. A few more improvements

I took some shortcuts in the flake definitions up to this point, just to keep it simple. Normally a flake has both a `packages` section and an `apps` section. The `apps` section is where we specify executable programs. If there is no `apps` section, then `nix run` will default to using the package, but that's not ideal.

A typical flake might have multiple packages and multiple apps. (We could even have multiple development environments.) Normally we would specify a default package and a default app. The command `nix run ` flakeurl#appname will run the app named appname from the `apps` section of *flakeurl*. If we don't specify *appname*, the default app is run.

To define an app, we specify the type and the path to the executable. We can re-use the definition from the `packages` section as shown below.

```
hello = {
  type = "app";
  program = pkgs.lib.getExe self.packages.${system}.hello;
};
```

Later we might want to add overlays or some configuration options to `nixpkgs` in our flake. We can include the scaffolding for it with the following change.

```
nixpkgsFor = forEachSystem (system: import nixpkgs {
  inherit system;
  config = { };
  overlays = [ ];
});
```

The Nix manual has more information on config options and overlays.

Putting everything together, we have:

flake.nix (version 6)

```
{
  description = "what does the cow say";

  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs";
  };

  outputs = { self, nixpkgs }:
    let

      supportedSystems = [ "x86_64-linux" "aarch64-linux" ];

      forEachSystem = nixpkgs.lib.genAttrs supportedSystems;

```

```
15      nixpkgsFor = forEachSystem (system: import nixpkgs {
16        inherit system;
17        config = { };
18        overlays = [ ];
19      });
20
21    in {
22
23      devShells = forEachSystem (system:
24        let pkgs = nixpkgsFor.${system}; in {
25          default = pkgs.mkShell {
26              packages = [ pkgs.cowsay ];
27          };
28      });
29
30      packages = forEachSystem (system:
31        let pkgs = nixpkgsFor.${system}; in rec {
32          hello = pkgs.stdenv.mkDerivation {
33            name = "cow-hello.sh";
34            src = ./.;
35            unpackPhase = "true";
36            buildPhase = ":";
37            installPhase =
38              ''
39                mkdir -p $out/bin
40                cp $src/cow-hello.sh $out/bin
41                chmod +x $out/bin/cow-hello.sh
42
43                # modify the cow-hello.sh script so it can find cowsay
44                COWSAY=$(type -p cowsay)
45                sed "s_cowsay_"$COWSAY"_" --in-place $out/bin/cow-hello.sh
46              '';
47            buildInputs = [ pkgs.cowsay ];
48          }; # mkDerivation
49
50          default = hello;
51      }); # packages
52
53      apps = forEachSystem (system:
54        let pkgs = nixpkgsFor.${system}; in rec {
55          hello = {
56            type = "app";
57            program = pkgs.lib.getExe self.packages.${system}.hello;
58          };
59
60          default = hello;
61      });
62    }; # outputs
63 }
```

Let's verify that it runs on our system.

```
$ git commit -am "refactored the flake"
[master 86a45c7] refactored the flake
 1 file changed, 22 insertions(+), 4 deletions(-)
$ nix run
evaluation warning: getExe: Package "cow-hello.sh" does not have the meta.mainProgram
attribute. We'll assume that the main program has the same name for now, but this
behavior is deprecated, because it leads to surprising errors when the assumption does
not hold. If the package has a main program, please set `meta.mainProgram` in its
definition to make this warning go away. Otherwise, if the package does not have a
main program, or if you don't control its definition, use getExe' to specify the name
to the program, such as lib.getExe' foo "bar".
this derivation will be built:
  /nix/store/ma7wfpsxwng701alypcv7sm8dlia9ac9-cow-hello.sh.drv
building '/nix/store/ma7wfpsxwng701alypcv7sm8dlia9ac9-cow-hello.sh.drv'...
 ________________________
< Hello from your flake! >
 ------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

9.2. Haskell

Start with an empty directory and create a git repository.

```
$ mkdir hello-haskell
$ cd hello-haskell
$ git init
Initialized empty Git repository in /home/amy/codeberg/nix-book/source/new-
flake/haskell-flake/hello-haskell/.git/
```

9.2.1. A simple Haskell program

Next, we'll create a simple Haskell program.

Main.hs

```haskell
import Network.HostName

main :: IO ()
main = do
  putStrLn "Hello from Haskell inside a Nix flake!"
  h <- getHostName
```

```
7   putStrLn $ "Your hostname is: " ++ h
```

9.2.2. Running the program manually (optional)

In this section you will learn how to do some development tasks manually, the "hard" way. This can help you understand the distinction between Nix's role and the Haskell build system you've chosen. Also, if you have a build problem but you're not sure if the fault is in your flake definition or some other configuration file, these commands can help you narrow it down. But you may wish to skip to the next section and come back here later.

Before we package the program, let's verify that it runs. We're going to need Haskell. By now you've probably figured out that we can write a `flake.nix` and define a development shell that includes Haskell. We'll do that shortly, but first I want to show you a handy shortcut. We can launch a *temporary* shell with any Nix packages we want. This is convenient when you just want to try out some new software and you're not sure if you'll use it again. It's also convenient when you're not ready to write `flake.nix` (perhaps you're not sure what tools and packages you need), and you want to experiment a bit first.

The command to enter a temporary shell is

```
nix shell -p installables
```

Where *installables* are flakes and other types of packages that you need. (You can learn more about these in the Nix manual.)

Some unsuitable shells

In this section, we will try commands that fail in subtle ways. Examining these failures will give you a much better understanding of Haskell development with Nix, and help you avoid (or at least diagnose) similar problems in future. If you're impatient, you can skip to the next section to see the right way to do it. You can come back to this section later to learn more.

Let's enter a shell with the Glasgow Haskell Compiler ("ghc") and try to run the program.

```
$ nix shell nixpkgs#ghc
$ runghc Main.hs     # Fails
Main.hs:1:1: error: [GHC-87110]
    Could not find module ‘Network.HostName’.
    Use :set -v to see a list of the files searched for.
  |
1 | import Network.HostName
  | ^^^^^^^^^^^^^^^^^^^^^^^
```

The error message tells us that we need the module `Network.HostName`. That module is provided by the Haskell package called `hostname`. Let's exit that shell and try again, this time adding the `hostname`

package.

```
$ exit
$ nix shell nixpkgs#ghc nixpkgs#hostname
$ runghc Main.hs     # Fails
Main.hs:1:1: error: [GHC-87110]
    Could not find module ‘Network.HostName’.
    Use :set -v to see a list of the files searched for.
  |
1 | import Network.HostName
  | ^^^^^^^^^^^^^^^^^^^^^^^
```

That reason that failed is that we asked for the wrong package. The Nix package `hostname` isn't the Haskell package we wanted, it's a different package entirely (an alias for `hostname-net-tools`.) The package we want is in the *package set* called `haskellPackages`, so we can refer to it as `haskellPackages.hostname`.

Let's try that again, with the correct package.

```
$ exit
$ nix shell nixpkgs#ghc nixpkgs#haskellPackages.hostname
these 2 paths will be fetched (0.04 MiB download, 0.17 MiB unpacked):
  /nix/store/hj4xz5hpkamgqsygr5zd9940mf9x2g7h-hostname-1.0
  /nix/store/6fc0wz0xj2lhyz541wfp25y95mlp01a4-hostname-1.0-doc
copying path '/nix/store/6fc0wz0xj2lhyz541wfp25y95mlp01a4-hostname-1.0-doc' from
'https://cache.nixos.org'...
copying path '/nix/store/hj4xz5hpkamgqsygr5zd9940mf9x2g7h-hostname-1.0' from
'https://cache.nixos.org'...
$ runghc Main.hs     # Fails
Main.hs:1:1: error: [GHC-87110]
    Could not find module ‘Network.HostName’.
    Use :set -v to see a list of the files searched for.
  |
1 | import Network.HostName
  | ^^^^^^^^^^^^^^^^^^^^^^^
```

Now what's wrong? The syntax we used in the `nix shell` command above is fine, but it doesn't make the package *available to GHC*!

A suitable shell for a quick test

Now we will create a shell that can run the program. When you need support for both a language and some of its packages, it's best to use one of the Nix functions that are specific to the programming language and build system. For Haskell, we can use the `ghcWithPackages` function. The command below is rather complex, and a complete explanation would be rather lengthy.

```
$ nix shell --impure --expr 'with import <nixpkgs> {}; haskellPackages.ghcWithPackages
(p: [ p.hostname ])'
```

```
this derivation will be built:
  /nix/store/0wbswdgsij0vf33bs1bb7mdz9fw37nkh-ghc-9.10.3-with-packages.drv
building '/nix/store/0wbswdgsij0vf33bs1bb7mdz9fw37nkh-ghc-9.10.3-with-packages.drv'...
$ runghc Main.hs
Hello from Haskell inside a Nix flake!
Your hostname is: wombat11k
```

Success! Now we know the program works.

9.2.3. The cabal file

It's time to write a Cabal file for this program. This is just an ordinary Cabal file; we don't need to do anything special for Nix.

hello-flake-haskell.cabal

```
cabal-version:   3.0
name:            hello-flake-haskell
version:         1.0.0
synopsis:        A simple demonstration using a Nix flake to package a Haskell program.
description:
  For more information and a tutorial on how to use this package,
  please see the README at <https://codeberg.org/mhwombat/hello-flake-haskell#readme>.
homepage:        https://codeberg.org/mhwombat/nix-book
bug-reports:     https://codeberg.org/mhwombat/nix-book/issues
license:         GPL-3.0-only
license-file:    LICENSE
author:          Amy de Buitléir
maintainer:      amy@nualeargais.ie
copyright:       (c) 2023 Amy de Buitléir
category:        Text
build-type:      Simple

executable hello-flake-haskell
  main-is:          Main.hs
  build-depends:
    base,
    hostname
    -- NOTE: Best practice is to specify version constraints for the packages we depend on.
    -- However, I'm confident that this package will only be used as a Nix flake.
    -- Nix will automatically ensure that anyone running this program is using the
    -- same library versions that I used to build it.
  default-language: Haskell2010
```

9.2.4. Building the program manually (optional)

In this section you will learn how to do some development tasks manually, the "hard" way. This can help you understand the distinction between Nix's role and the Haskell build system you've chosen. Also, if you have a build problem but you're not sure if the fault is in your flake definition or some other configuration file, these commands can help you narrow it down. But you may wish to skip to the next section and come back here later.

We won't write `flake.nix` just yet. First we'll try building the package manually. (If you didn't run the `nix shell` command from earlier, do so now.)

```
$ cabal build
sh: line 42: cabal: command not found
```

The `cabal` command is provided by the `cabal-install` package. The error happens because we don't have `cabal-install` available in the temporary shell. We can correct that.

```
$ nix shell --impure --expr 'with import <nixpkgs> {}; haskellPackages.ghcWithPackages
(p: [ p.hostname p.cabal-install ])'
this derivation will be built:
  /nix/store/d0kcssjwwwbjah1v097srkbj3808pfgw-ghc-9.10.3-with-packages.drv
these 24 paths will be fetched (50.39 MiB download, 768.77 MiB unpacked):
  /nix/store/c96p2zbws4qsi7y032z3rxwpcnsmxa0n-Cabal-3.16.0.0
  /nix/store/869p5aaksxd0h2gcz4ijsdwd5pn8khr9-Cabal-3.16.0.0-doc
  /nix/store/fn0qwls0jz63b6lr267xqr2cayxav6ll-Cabal-syntax-3.16.0.0
  /nix/store/lc5fixlrkn9vjly2qz6vp82j60qx7dmc-Cabal-syntax-3.16.0.0-doc
  /nix/store/kdwghzkfpwl5v7y780fn4m8418x5pj6v-HTTP-4000.4.1
  /nix/store/qcn5m2fdiv5wd1vhsim4nqfw58k3i359-HTTP-4000.4.1-doc
  /nix/store/s8w1zrslahvah80cmkdqv7vk5qgpip1w-cabal-install-3.16.0.0
  /nix/store/10v0c4bxlm7swlx4ypvqw5akbfbzk9c4-cabal-install-3.16.0.0-doc
  /nix/store/4r8h00nf3zsg7ksmxikl6lkrn4dc6ajg-cabal-install-solver-3.16.0.0
  /nix/store/rwlbg49q1c3fhdg48lgb52xhp0bkp80c-cabal-install-solver-3.16.0.0-doc
  /nix/store/vmsin8v6jw9fwfw05151agabxx3flkjj-echo-0.1.4
  /nix/store/xj4sj93vwscmmy3d61w0nxg2q6qxfg6l-echo-0.1.4-doc
  /nix/store/vlsxf1137zr6fps00bw6hxzhlym3n2ws-ed25519-0.0.5.0
  /nix/store/8gb0d2lpy5pjngdpfqlhbnrcv3jckv98-ed25519-0.0.5.0-doc
  /nix/store/gizkjbpg30w5zbh76dr19rw43fl085s7-edit-distance-0.2.2.1
  /nix/store/c35yhzzjlh6wfwrhb5wrdzdm4zgqibm2-edit-distance-0.2.2.1-doc
  /nix/store/2qmnj9cghfwhdyg8m89f24gyw802vcal-hackage-security-0.6.3.2
  /nix/store/912psk9lh3nb74h9bax3yqrdqx2m0ma8-hackage-security-0.6.3.2-doc
  /nix/store/ni9dbpzghnx064mi6v67ifdlx38pk95b-open-browser-0.4.0.0
  /nix/store/aw8nqiw3isils1d8v3wb65dk97qsynb3-open-browser-0.4.0.0-doc
  /nix/store/mmcsvzf7asnj0pgnhvjs5yg9msr058pf-regex-posix-0.96.0.2
  /nix/store/wvgycws500wfsawq7ja6nx2692c1zvqv-regex-posix-0.96.0.2-doc
  /nix/store/lz9lk15whi3cfb402zxysja7qmbw63f1-resolv-0.2.0.3
  /nix/store/a14fzjy1wq53kbwjnphrv0dwjry4mmf4-resolv-0.2.0.3-doc
copying path '/nix/store/lc5fixlrkn9vjly2qz6vp82j60qx7dmc-Cabal-syntax-3.16.0.0-doc'
```

```
from 'https://cache.nixos.org'...
copying path '/nix/store/xj4sj93vwscmmy3d61w0nxg2q6qxfg6l-echo-0.1.4-doc' from
'https://cache.nixos.org'...
copying path '/nix/store/8gb0d2lpy5pjngdpfqlhbnrcv3jckv98-ed25519-0.0.5.0-doc' from
'https://cache.nixos.org'...
copying path '/nix/store/c35yhzzjlh6wfwrhb5wrdzdm4zgqibm2-edit-distance-0.2.2.1-doc'
from 'https://cache.nixos.org'...
copying path '/nix/store/qcn5m2fdiv5wd1vhsim4nqfw58k3i359-HTTP-4000.4.1-doc' from
'https://cache.nixos.org'...
copying path '/nix/store/aw8nqiw3isils1d8v3wb65dk97qsynb3-open-browser-0.4.0.0-doc'
from 'https://cache.nixos.org'...
copying path '/nix/store/wvgycws500wfsawq7ja6nx2692c1zvqv-regex-posix-0.96.0.2-doc'
from 'https://cache.nixos.org'...
copying path '/nix/store/a14fzjy1wq53kbwjnphrv0dwjry4mmf4-resolv-0.2.0.3-doc' from
'https://cache.nixos.org'...
copying path '/nix/store/vmsin8v6jw9fwfw05151agabxx3flkjj-echo-0.1.4' from
'https://cache.nixos.org'...
copying path '/nix/store/vlsxf1137zr6fps00bw6hxzhlym3n2ws-ed25519-0.0.5.0' from
'https://cache.nixos.org'...
copying path '/nix/store/ni9dbpzghnx064mi6v67ifdlx38pk95b-open-browser-0.4.0.0' from
'https://cache.nixos.org'...
copying path '/nix/store/gizkjbpg30w5zbh76dr19rw43fl085s7-edit-distance-0.2.2.1' from
'https://cache.nixos.org'...
copying path '/nix/store/mmcsvzf7asnj0pgnhvjs5yg9msr058pf-regex-posix-0.96.0.2' from
'https://cache.nixos.org'...
copying path '/nix/store/lz9lk15whi3cfb402zxysja7qmbw63f1-resolv-0.2.0.3' from
'https://cache.nixos.org'...
copying path '/nix/store/kdwghzkfpwl5v7y780fn4m8418x5pj6v-HTTP-4000.4.1' from
'https://cache.nixos.org'...
copying path '/nix/store/869p5aaksxd0h2gcz4ijsdwd5pn8khr9-Cabal-3.16.0.0-doc' from
'https://cache.nixos.org'...
copying path '/nix/store/fn0qwls0jz63b6lr267xqr2cayxav6ll-Cabal-syntax-3.16.0.0' from
'https://cache.nixos.org'...
copying path '/nix/store/912psk9lh3nb74h9bax3yqrdqx2m0ma8-hackage-security-0.6.3.2-
doc' from 'https://cache.nixos.org'...
copying path '/nix/store/rwlbg49q1c3fhdg48lgb52xhp0bkp80c-cabal-install-solver-
3.16.0.0-doc' from 'https://cache.nixos.org'...
copying path '/nix/store/10v0c4bxlm7swlx4ypvqw5akbfbzk9c4-cabal-install-3.16.0.0-doc'
from 'https://cache.nixos.org'...
copying path '/nix/store/c96p2zbws4qsi7y032z3rxwpcnsmxa0n-Cabal-3.16.0.0' from
'https://cache.nixos.org'...
copying path '/nix/store/4r8h00nf3zsg7ksmxikl6lkrn4dc6ajg-cabal-install-solver-
3.16.0.0' from 'https://cache.nixos.org'...
copying path '/nix/store/2qmnj9cghfwhdyg8m89f24gyw802vcal-hackage-security-0.6.3.2'
from 'https://cache.nixos.org'...
copying path '/nix/store/s8w1zrslahvah80cmkdqv7vk5qgpip1w-cabal-install-3.16.0.0' from
'https://cache.nixos.org'...
building '/nix/store/d0kcssjwwwbjah1v097srkbj3808pfgw-ghc-9.10.3-with-packages.drv'...
```

Note that we're now inside a temporary shell inside the previous temporary shell! To get back to

the original shell, we have to `exit` twice. Alternatively, we could have done `exit` followed by the second `nix-shell` command.

```
$ cabal build
. . .
Preprocessing executable 'hello-flake-haskell' for hello-flake-haskell-1.0.0...
Building executable 'hello-flake-haskell' for hello-flake-haskell-1.0.0...
[1 of 1] Compiling Main             ( Main.hs, dist-newstyle/build/x86_64-linux/ghc-
9.10.3/hello-flake-haskell-1.0.0/x/hello-flake-haskell/build/hello-flake-
haskell/hello-flake-haskell-tmp/Main.o )
[2 of 2] Linking dist-newstyle/build/x86_64-linux/ghc-9.10.3/hello-flake-haskell-
1.0.0/x/hello-flake-haskell/build/hello-flake-haskell/hello-flake-haskell
```

After a lot of output messages, the build succeeds.

9.2.5. The Nix flake

Now we should write `flake.nix`. We already know how to write most of the flake from the examples we did earlier. The two parts that will be different are the development shell and the package builder.

However, there's a *much* simpler way, using `haskell-flake`. The following command will create `flake.nix` based on their template.

```
$ nix flake init -t github:srid/haskell-flake
wrote: "/home/amy/codeberg/nix-book/source/new-flake/haskell-flake/hello-
haskell/flake.nix"
```

Examining the flake, you'll notice that it is well-commented. The only thing we need to change for now is the name in `packages.default`; I've highlighted the change below.

flake.nix

```
{
  inputs = {
    nixpkgs.url = "github:nixos/nixpkgs/nixpkgs-unstable";
    flake-parts.url = "github:hercules-ci/flake-parts";
    haskell-flake.url = "github:srid/haskell-flake";
  };
  outputs = inputs@{ self, nixpkgs, flake-parts, ... }:
    flake-parts.lib.mkFlake { inherit inputs; } {
      systems = nixpkgs.lib.systems.flakeExposed;
      imports = [ inputs.haskell-flake.flakeModule ];

      perSystem = { self', pkgs, ... }: {

        # Typically, you just want a single project named "default". But
        # multiple projects are also possible, each using different GHC version.
```

```
16        haskellProjects.default = {
17          # The base package set representing a specific GHC version.
18          # By default, this is pkgs.haskellPackages.
19          # You may also create your own. See
    https://community.flake.parts/haskell-flake/package-set
20          # basePackages = pkgs.haskellPackages;
21
22          # Extra package information. See https://community.flake.parts/haskell-
    flake/dependency
23          #
24          # Note that local packages are automatically included in `packages`
25          # (defined by `defaults.packages` option).
26          #
27          packages = {
28            # aeson.source = "1.5.0.0";      # Override aeson to a custom version
    from Hackage
29            # shower.source = inputs.shower; # Override shower to a custom source
    path
30          };
31          settings = {
32            #  aeson = {
33            #    check = false;
34            #  };
35            #  relude = {
36            #    haddock = false;
37            #    broken = false;
38            #  };
39          };
40
41          devShell = {
42            # Enabled by default
43            # enable = true;
44
45            # Programs you want to make available in the shell.
46            # Default programs can be disabled by setting to 'null'
47            # tools = hp: { fourmolu = hp.fourmolu; ghcid = null; };
48
49            # Check that haskell-language-server works
50            # hlsCheck.enable = true; # Requires sandbox to be disabled
51          };
52        };
53
54        # haskell-flake doesn't set the default package, but you can do it here.
55        packages.default = self'.packages.hello-flake-haskell;
56      };
57    };
58 }
```

We also need a `LICENSE` file. For now, this can be empty.

```
$ touch LICENSE
```

9.2.6. Building the program

Let's try out the new flake. Nix flakes only "see" files that are part of the repository. We need to add all of the important files to the repo before building or running the flake.

```
$ git add flake.nix hello-flake-haskell.cabal LICENSE Main.hs
$ nix build
warning: Git tree '/home/amy/codeberg/nix-book/source/new-flake/haskell-flake/hello-
haskell' is dirty
unpacking 'github:nixos/nixpkgs/4533d9293756b63904b7238acb84ac8fe4c8c2c4' into the Git
cache...
warning: creating lock file "/home/amy/codeberg/nix-book/source/new-flake/haskell-
flake/hello-haskell/flake.lock":
• Added input 'flake-parts':
    'github:hercules-ci/flake-
parts/57928607ea566b5db3ad13af0e57e921e6b12381?narHash=sha256-
AnYjnFWgS49RlqX7LrC4uA%2BsCCDBj0Ry/WOJ5XWAsa0%3D' (2026-02-02)
• Added input 'flake-parts/nixpkgs-lib':
    'github:nix-
community/nixpkgs.lib/72716169fe93074c333e8d0173151350670b824c?narHash=sha256-
cBEymOf4/o3FD5AZnzC3J9hLbiZ%2BQDT/KDuyHXVJOpM%3D' (2026-02-01)
• Added input 'haskell-flake':
    'github:srid/haskell-
flake/d5c7dfaa97dbb446a615923ad7e4d9efabe73c52?narHash=sha256-
Z7cC5jPmLYrEEzE%2BWNU/NIgwCLa5P4W5b9uXzx%2BHwkM%3D' (2026-02-01)
• Added input 'nixpkgs':
    'github:nixos/nixpkgs/4533d9293756b63904b7238acb84ac8fe4c8c2c4?narHash=sha256-
tbS0Ebx2PiA1FRW8mt8oejR0qMXmziJmPaU1d4kYY9g%3D' (2026-02-03)
building '/nix/store/6vhawjab6znm7lgnrdff0s7nwdjimcdn-cabal2nix-hello-flake-
haskell.drv'...
these 2 derivations will be built:
  /nix/store/13xixgzz1nxg8gzi5j07xc5mz7rwq8gl-hello-flake-haskell-source-1.0.0.drv
  /nix/store/45m9kl6xk06h9n8dxig8ajclwsarcv6r-hello-flake-haskell-1.0.0.drv
these 2 paths will be fetched (0.04 MiB download, 0.17 MiB unpacked):
  /nix/store/25vyn1axss60sfl6kyl29781zqf49vj3-hostname-1.0
  /nix/store/kyyqjc2244a6qs7ic8yaaad15xp47k7y-hostname-1.0-doc
copying path '/nix/store/kyyqjc2244a6qs7ic8yaaad15xp47k7y-hostname-1.0-doc' from
'https://cache.nixos.org'...
copying path '/nix/store/25vyn1axss60sfl6kyl29781zqf49vj3-hostname-1.0' from
'https://cache.nixos.org'...
building '/nix/store/13xixgzz1nxg8gzi5j07xc5mz7rwq8gl-hello-flake-haskell-source-
1.0.0.drv'...
building '/nix/store/45m9kl6xk06h9n8dxig8ajclwsarcv6r-hello-flake-haskell-
1.0.0.drv'...
```

We'll deal with those warnings later. The important thing for now is that the build succeeded.

9.2.7. Running the program

Now we can run the program.

```
$ nix run
warning: Git tree '/home/amy/codeberg/nix-book/source/new-flake/haskell-flake/hello-
haskell' is dirty
these 2 derivations will be built:
  /nix/store/vq0rxqb0vcmcbfiswyh8qligm9bpd7n0-hello-flake-haskell-source-1.0.0.drv
  /nix/store/rbir4wbzgpm097037icn5yipqifm7v4n-hello-flake-haskell-1.0.0.drv
building '/nix/store/vq0rxqb0vcmcbfiswyh8qligm9bpd7n0-hello-flake-haskell-source-
1.0.0.drv'...
building '/nix/store/rbir4wbzgpm097037icn5yipqifm7v4n-hello-flake-haskell-
1.0.0.drv'...
Hello from Haskell inside a Nix flake!
Your hostname is: wombat11k
```

By the way, we didn't need to do `nix build` earlier. The `nix run` command will first build the program for us if needed.

We'd like to share this package with others, but first we should do some cleanup. It's time to deal with those warnings. When the package was built, Nix created a `flake.lock` file. We need to add this to the repo, and commit all important files.

```
$ git add flake.lock
$ git commit -a -m 'initial commit'
[master (root-commit) 3dd4794] initial commit
 5 files changed, 170 insertions(+)
 create mode 100644 LICENSE
 create mode 100644 Main.hs
 create mode 100644 flake.lock
 create mode 100644 flake.nix
 create mode 100644 hello-flake-haskell.cabal
```

You can test that your package is properly configured by going to another directory and running it from there.

```
$ cd ..
$ nix run ./hello-haskell
Hello from Haskell inside a Nix flake!
Your hostname is: wombat11k
```

If you move the project to a public repo, anyone can run it. Recall from the beginning of the tutorial that you were able to run `hello-flake` directly from my repo with the following command.

```
nix run "git+https://codeberg.org/mhwombat/hello-flake"
```

Modify the URL accordingly and invite someone else to run your new Haskell flake.

9.3. Python

Start with an empty directory and create a git repository.

```
$ mkdir hello-python
$ cd hello-python
$ git init
Initialized empty Git repository in /home/amy/codeberg/nix-book/source/new-
flake/python-flake/hello-python/.git/
```

9.3.1. A simple Python program

Next, we'll create a simple Python program.

hello.py

```
1 #!/usr/bin/env python
2
3 def main():
4     print("Hello from inside a Python program built with a Nix flake!")
5
6 if __name__ == "__main__":
7     main()
```

9.3.2. Running the program manually (optional)

In this section you will learn how to do some development tasks manually, the "hard" way. This can help you understand the distinction between Nix's role and the Python build system you've chosen. Also, if you have a build problem but you're not sure if the fault is in your flake definition or some other configuration file, these commands can help you narrow it down. But you may wish to skip to the next section and come back here later.

Before we package the program, let's verify that it runs. We're going to need Python. By now you've probably figured out that we can write a `flake.nix` and define a development shell that includes Python. We'll do that shortly, but first I want to show you a handy shortcut. We can launch a *temporary* shell with any Nix packages we want. This is convenient when you just want to try out some new software and you're not sure if you'll use it again. It's also convenient when you're not ready to write `flake.nix` (perhaps you're not sure what tools and packages you need), and you want to experiment a bit first.

The command to enter a temporary shell is

```
nix shell -p installables
```

Where *installables* are flakes and other types of packages that you need. (You can learn more about these in the Nix manual.)

Let's enter a shell with Python so we can test the program.

```
$ nix shell nixpkgs#python3
$ python hello.py
Hello from inside a Python program built with a Nix flake!
```

9.3.3. Configuring setuptools

Next, configure the package. We'll use Python's setuptools, but you can use other build tools. For more information on setuptools, see the Setuptools documentation, especially the section on pyproject.toml.

pyproject.toml

```
[project]
name = "hello"
version = "0.1.0"
dependencies = [ ]

[build-system]
requires = ["setuptools"]
build-backend = "setuptools.build_meta"

[project.scripts]
hello = "hello:main"
```

9.3.4. Building the program manually (optional)

In this section you will learn how to do some development tasks manually, the "hard" way. This can help you understand the distinction between Nix's role and the Python build system you've chosen. Also, if you have a build problem but you're not sure if the fault is in your flake definition or some other configuration file, these commands can help you narrow it down. But you may wish to skip to the next section and come back here later.

We won't write `flake.nix` just yet. First we'll try building the package manually. (If you didn't run the `nix shell` command from earlier, do so now.)

```
$ python -m build     # Fails
/nix/store/qzc04a3npl70cyyy6flnnrb2ig3kayxm-python3-3.13.11/bin/python: No module
named build
```

The missing module error happens because we don't have `build` available in the temporary shell.

We can fix that by adding “build” to the temporary shell. When you need support for both a language and some of its packages, it’s best to use one of the Nix functions that are specific to the programming language and build system. For Python, we can use the `withPackages` function.

```
$ nix-shell -p "python3.withPackages (ps: with ps; [ build ])"
these 2 derivations will be built:
  /nix/store/hvxxm22ddlig6lvn8pifxa7kpb67acq6-builder.pl.drv
  /nix/store/ihs8qw69nqk80428xifsdz6vqyg0n19h-python3-3.13.11-env.drv
these 4 paths will be fetched (0.06 MiB download, 0.34 MiB unpacked):
  /nix/store/1yvbdnys10qw6y0339vaki40wfc3vw4j-bin
  /nix/store/6r8hv0jpa2n34i2lxknsz6mgzgxwdg4j-make-binary-wrapper-hook
  /nix/store/rjs7qfglhi7cgd3sbvgqhl6vajps3724-python3.13-build-1.3.0
  /nix/store/rrbrry1vjvvf9vsqghq578s1bm5fy1sn-python3.13-pyproject-hooks-1.2.0
copying path '/nix/store/1yvbdnys10qw6y0339vaki40wfc3vw4j-bin' from
'https://cache.nixos.org'...
copying path '/nix/store/6r8hv0jpa2n34i2lxknsz6mgzgxwdg4j-make-binary-wrapper-hook'
from 'https://cache.nixos.org'...
copying path '/nix/store/rrbrry1vjvvf9vsqghq578s1bm5fy1sn-python3.13-pyproject-hooks-
1.2.0' from 'https://cache.nixos.org'...
building '/nix/store/hvxxm22ddlig6lvn8pifxa7kpb67acq6-builder.pl.drv'...
Running phase: patchPhase
copying path '/nix/store/rjs7qfglhi7cgd3sbvgqhl6vajps3724-python3.13-build-1.3.0' from
'https://cache.nixos.org'...
Running phase: updateAutotoolsGnuConfigScriptsPhase
Running phase: configurePhase
no configure script, doing nothing
Running phase: buildPhase
Running phase: checkPhase
Running phase: installPhase
no Makefile or custom installPhase, doing nothing
Running phase: fixupPhase
shrinking RPATHs of ELF executables and libraries in
/nix/store/119wcss8hpblzq13acpgivaiy5vg3hv8-builder.pl
checking for references to /build/ in /nix/store/119wcss8hpblzq13acpgivaiy5vg3hv8-
builder.pl...
patching script interpreter paths in /nix/store/119wcss8hpblzq13acpgivaiy5vg3hv8-
builder.pl
building '/nix/store/ihs8qw69nqk80428xifsdz6vqyg0n19h-python3-3.13.11-env.drv'...
created 223 symlinks in user environment
```

Note that we’re now inside a temporary shell inside the previous temporary shell! To get back to the original shell, we have to `exit` twice. Alternatively, we could have done `exit` followed by the second `nix-shell` command.

```
$ python -m build
```

After a lot of output messages, the build succeeds.

9.3.5. The Nix flake

Now we should write `flake.nix`. We already know how to write most of the flake from the examples we did earlier. The two parts that will be different are the development shell and the package builder.

Let's start with the development shell. It seems logical to write something like the following.

```
      devShells = forAllSupportedSystems (system:
        let
          pkgs = nixpkgsFor.${system};
          pythonEnv = pkgs.python3.withPackages (ps: [ ps.build ]);
        in {
          default = pkgs.mkShell {
              packages = [ pythonEnv ];
          };
      });
```

Note that we need the parentheses to prevent `python.withPackages` and the argument from being processed as two separate tokens. Suppose we wanted to work with `virtualenv` and `pip` instead of `build`. We could write something like the following.

```
      devShells = forAllSupportedSystems (system:
        let
          pkgs = nixpkgsFor.${system};
          pythonEnv = pkgs.python3.withPackages (ps: [ ps.virtualenv ps.pip ]);
        in {
          default = pkgs.mkShell {
              packages = [ pythonEnv ];
          };
      });
```

For the package builder, we can use the `buildPythonApplication` function.

```
       packages = forAllSupportedSystems (system:
        let pkgs = nixpkgsFor.${system}; in rec {
          hello-flake-python = pkgs.python3Packages.buildPythonApplication {
            name = "hello-flake-python";
            pyproject = true;
            src = ./.;
            build-system = with pkgs.python3Packages; [ setuptools ];
          };
      });
```

If you put all the pieces together, your `flake.nix` should look something like this.

flake.nix

```nix
{
  description = "a very simple and friendly flake written in Python";

  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs";
  };

  outputs = { self, nixpkgs }:
    let
      supportedSystems = nixpkgs.lib.systems.flakeExposed;

      forAllSupportedSystems = nixpkgs.lib.genAttrs supportedSystems;

      nixpkgsFor = forAllSupportedSystems (system: import nixpkgs {
        inherit system;
        config = { };
        overlays = [ ];
      });

    in {
      devShells = forAllSupportedSystems (system:
        let
          pkgs = nixpkgsFor.${system};
          pythonEnv = pkgs.python3.withPackages (ps: [ ps.build ]);
        in {
          default = pkgs.mkShell {
              packages = [ pythonEnv ];
          };
      });

       packages = forAllSupportedSystems (system:
        let pkgs = nixpkgsFor.${system}; in rec {
          hello-flake-python = pkgs.python3Packages.buildPythonApplication {
            name = "hello-flake-python";
            pyproject = true;
            src = ./.;
            build-system = with pkgs.python3Packages; [ setuptools ];
          };
      });

      apps = forAllSupportedSystems (system:
        let pkgs = nixpkgsFor.${system}; in rec {
          hello-flake-python = {
            type = "app";
            program = pkgs.lib.getExe' self.packages.${system}.hello-flake-python
"hello";
          };

          default = hello-flake-python;
```

```
49       });
50
51     };
52 }
```

9.3.6. Building the program

Let's try out the new flake.

```
$ nix build     # Fails
error: Path 'flake.nix' in the repository "/home/amy/codeberg/nix-book/source/new-
flake/python-flake/hello-python" is not tracked by Git.

       To make it visible to Nix, run:

       git -C "/home/amy/codeberg/nix-book/source/new-flake/python-flake/hello-python"
add "flake.nix"
```

Nix flakes only "see" files that are part of the repository. We need to add all of the important files to the repo before building or running the flake.

```
$ git add flake.nix setup.py hello.py
fatal: pathspec 'setup.py' did not match any files
$ nix build
error: Path 'flake.nix' in the repository "/home/amy/codeberg/nix-book/source/new-
flake/python-flake/hello-python" is not tracked by Git.

       To make it visible to Nix, run:

       git -C "/home/amy/codeberg/nix-book/source/new-flake/python-flake/hello-python"
add "flake.nix"
```

We'll deal with those warnings later. The important thing for now is that the build succeeded.

9.3.7. Running the program

Now we can run the program.

```
$ nix run
error: Path 'flake.nix' in the repository "/home/amy/codeberg/nix-book/source/new-
flake/python-flake/hello-python" is not tracked by Git.

       To make it visible to Nix, run:

       git -C "/home/amy/codeberg/nix-book/source/new-flake/python-flake/hello-python"
add "flake.nix"
```

By the way, we didn't need to do `nix build` earlier. The `nix run` command will first build the program for us if needed.

We'd like to share this package with others, but first we should do some cleanup. It's time to deal with those warnings. When the package was built, Nix created a `flake.lock` file. We need to add this to the repo, and commit all important files.

```
$ git add flake.lock
fatal: pathspec 'flake.lock' did not match any files
$ git commit -a -m 'initial commit'
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    dist/
    flake.nix
    hello.egg-info/
    hello.py
    pyproject.toml

nothing added to commit but untracked files present (use "git add" to track)
```

You can test that your package is properly configured by going to another directory and running it from there.

```
$ cd ..
$ nix run ./hello-python
error: Path 'flake.nix' in the repository "/home/amy/codeberg/nix-book/source/new-
flake/python-flake/hello-python" is not tracked by Git.

       To make it visible to Nix, run:

       git -C "/home/amy/codeberg/nix-book/source/new-flake/python-flake/hello-python"
add "flake.nix"
```

If you move the project to a public repo, anyone can run it. Recall from the beginning of the tutorial that you were able to run `hello-flake` directly from my repo with the following command.

```
nix run "git+https://codeberg.org/mhwombat/hello-flake"
```

Modify the URL accordingly and invite someone else to run your new Python flake.

Chapter 10. Recipes

This chapter provides examples of how to use Nix in a variety of scenarios. Multiple types of recipes are provided are provided for some scenarios; comparing the different recipes will help you better understand Nix.

- *"Ad hoc" shells* are useful when you want to quickly create an environment for a one-off task.
- *Nix flakes* are the recommended approach for development projects, including defining environments that you will use more than once.

10.1. Running programs directly (without installing them)

10.1.1. Run a top level package from the Nixpkgs/NixOS repo

```
$ nix run nixpkgs#cowsay "Moo!"
this path will be fetched (0.01 MiB download, 0.05 MiB unpacked):
  /nix/store/y5awi2qqhx6rx6j7fh6ik2k8aagdyy1z-cowsay-3.8.4
copying path '/nix/store/y5awi2qqhx6rx6j7fh6ik2k8aagdyy1z-cowsay-3.8.4' from
'https://cache.nixos.org'...
 ______
< Moo! >
 ------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

10.1.2. Run a flake

Run a flake defined in a local file

```
$ nix run ~/codeberg/hello-flake
Hello from your flake!
```

Run a flake defined in a remote git repo

```
$ nix run git+https://codeberg.org/mhwombat/hello-flake
Hello from your flake!
```

To use a package from GitHub, GitLab, or any other public platform, modify the URL accordingly. To run a specific branch, use the command below.

```
nix run git+https://codeberg.org/mhwombat/hello-flake?ref=main
```

To run a specific branch and revision, use the command below.

```
nix run git+https://codeberg.org/mhwombat/hello-
flake?ref=main&rev=d44728bce88a6f9d1d37dbf4720ece455e997606
```

Run a flake defined in a zip archive

```
$ nix run https://codeberg.org/mhwombat/hello-flake/archive/main.zip
unpacking 'https://codeberg.org/mhwombat/hello-flake/archive/main.zip' into the Git
cache...
Hello from your flake!
```

Run a flake defined in a compressed tar archive

```
$ nix run https://codeberg.org/mhwombat/hello-flake/archive/main.tar.gz
unpacking 'https://codeberg.org/mhwombat/hello-flake/archive/main.tar.gz' into the Git
cache...
Hello from your flake!
```

Run other types of flake references

See https://nix.dev/manual/nix/stable/command-ref/new-cli/nix3-flake#flake-reference-attributes.

10.2. Ad hoc environments

10.2.1. Access a top level package from the Nixpkgs/NixOS repo

```
$ nix shell nixpkgs#cowsay
$ cowsay "moo"
 _____
< moo >
 -----
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

10.2.2. Access a flake

In this example, we will use a flake defined in a remote git repo. However, you can use any of the

flake reference styles defined in Section 10.1.2, “Run a flake”.

```
$ nix shell git+https://codeberg.org/mhwombat/hello-flake
$ hello-flake
Hello from your flake!
```

10.3. Scripts

You can use `nix shell` to run scripts in arbitrary languages, providing the necessary dependencies. This is particularly convenient for standalone scripts because you don’t need to create a repo and write a separate `flake.nix`. The script should start with two *"shebang"* (`#!`) commands. The first should invoke `nix`. The second should declares the script interpreter and any dependencies.

10.3.1. Access a top level package from the Nixpkgs/NixOS repo

Script

```
1 #! /usr/bin/env nix
2 #! nix shell nixpkgs#hello nixpkgs#cowsay --command bash
3 hello
4 cowsay "Pretty cool, huh?"
```

Output

```
these 2 paths will be fetched (0.06 MiB download, 0.27 MiB unpacked):
  /nix/store/6mk2k28339wgplwz7l32ilp66ldqlr6v-cowsay-3.8.4-man
  /nix/store/nkshga4h8jrz0r21l72k28axhllidp7p-hello-2.12.2
copying path '/nix/store/6mk2k28339wgplwz7l32ilp66ldqlr6v-cowsay-3.8.4-man' from
'https://cache.nixos.org'...
copying path '/nix/store/nkshga4h8jrz0r21l72k28axhllidp7p-hello-2.12.2' from
'https://cache.nixos.org'...
Hello, world!
 ___________________
< Pretty cool, huh? >
 -------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

10.3.2. Access a flake

In this example, we will use a flake defined in a remote git repo. However, you can use any of the flake reference styles defined in Section 10.1.2, “Run a flake”.

Script

```
#! /usr/bin/env nix
#! nix shell git+https://codeberg.org/mhwombat/hello-flake --command bash
hello-flake
```

Output

```
Hello from your flake!
```

10.3.3. Access a Haskell library package in the nixpkgs repo (without a `.cabal` file)

Occasionally you might want to run a short Haskell program that depends on a Haskell library, but you don't want to bother writing a cabal file.

Example: Access the `extra` package from the `haskellPackages` set in the nixpkgs repo.

Script

```
#! /usr/bin/env nix-shell
#! nix-shell -p "haskellPackages.ghcWithPackages (p: [p.extra])"
#! nix-shell -i runghc

import Data.List.Extra

main :: IO ()
main = do
  print $ lower "ABCDE"
  print $ upper "XYZ"
```

Output

```
this derivation will be built:
  /nix/store/6j1myl05cb10ais5zbf2q7fkc6ya19dz-ghc-9.10.3-with-packages.drv
these 4 paths will be fetched (0.03 MiB download, 0.09 MiB unpacked):
  /nix/store/h5m12gmjjxs0sa13scpcg6zrx73c7c6r-die-hook
  /nix/store/ddlxl7m20v4r1895gar03afjif68wa2l-lndir-1.0.5
  /nix/store/48zc854y65q0jvsa2na6liawgqvh69cq-make-shell-wrapper-hook
  /nix/store/wx5ivjgmzwl9222fpdy5h0c1ysa9cldn-stdenv-linux
copying path '/nix/store/h5m12gmjjxs0sa13scpcg6zrx73c7c6r-die-hook' from
'https://cache.nixos.org'...
copying path '/nix/store/ddlxl7m20v4r1895gar03afjif68wa2l-lndir-1.0.5' from
'https://cache.nixos.org'...
copying path '/nix/store/wx5ivjgmzwl9222fpdy5h0c1ysa9cldn-stdenv-linux' from
'https://cache.nixos.org'...
copying path '/nix/store/48zc854y65q0jvsa2na6liawgqvh69cq-make-shell-wrapper-hook'
from 'https://cache.nixos.org'...
building '/nix/store/6j1myl05cb10ais5zbf2q7fkc6ya19dz-ghc-9.10.3-with-packages.drv'...
```

```
Warning: include-dirs: /nix/store/2cs1kkhwmv3bx6wz8f7kp0v2h5a1liam-ghc-9.10.3-with-
packages/lib/ghc-9.10.3/lib/../lib/x86_64-linux-ghc-9.10.3/directory-1.3.8.5-
b115/include doesn't exist or isn't a directory
"abcde"
"XYZ"
```

10.3.4. Access a Python library package in the nixpkgs repo (without using a Python builder)

Occasionally you might want to run a short Python program that depends on a Python library, but you don't want to bother configuring a builder.

Example: Access the `html_sanitizer` package from the `python3nnPackages` set in the nixpkgs repo.

Script

```
#! /usr/bin/env nix-shell
#! nix-shell -i python3 -p python3Packages.html-sanitizer

from html_sanitizer import Sanitizer
sanitizer = Sanitizer()  # default configuration

original='<span style="font-weight:bold">some text</span>'
print('original: ', original)

sanitized=sanitizer.sanitize(original)
print('sanitized: ', sanitized)
```

Output

```
these 2 paths will be fetched (0.05 MiB download, 0.24 MiB unpacked):
  /nix/store/1jx83k5vks83jdhhyf9l284nyparh40j-python3.13-html-sanitizer-2.6
  /nix/store/b4bmvxwv6ddav98li9hb5jdj0pk53spl-python3.13-lxml-html-clean-0.4.3
copying path '/nix/store/b4bmvxwv6ddav98li9hb5jdj0pk53spl-python3.13-lxml-html-clean-
0.4.3' from 'https://cache.nixos.org'...
copying path '/nix/store/1jx83k5vks83jdhhyf9l284nyparh40j-python3.13-html-sanitizer-
2.6' from 'https://cache.nixos.org'...
original:  <span style="font-weight:bold">some text</span>
sanitized:  <strong>some text</strong>
```

10.4. Development environments

10.4.1. Access a top level package from the Nixpkgs/NixOS repo

flake.nix

```
{
  inputs = {
```

```
    nixpkgs.url = "github:NixOS/nixpkgs";
    flake-utils.url = "github:numtide/flake-utils";
  };

  outputs = { self, nixpkgs, flake-utils }:
    flake-utils.lib.eachDefaultSystem (system:
      let
        pkgs = import nixpkgs { inherit system; };
      in
      {
        devShells = rec {
          default = pkgs.mkShell {
            packages = [ pkgs.cowsay ];
          };
        };
      }
    );
}
```

Here's a demonstration using the shell.

```
$ cowsay "Moo!"     # Fails; dependency not available
bash: line 17: cowsay: command not found
$ nix develop
$ cowsay "Moo!"     # Works in development environment
 ______
< Moo! >
 ------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

10.4.2. Access a flake

In this example, we will use a flake defined in a remote git repo. However, you can use any of the flake reference styles defined in Section 10.1.2, "Run a flake".

flake.nix

```
{
  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs";
    flake-utils.url = "github:numtide/flake-utils";
    hello-flake.url = "git+https://codeberg.org/mhwombat/hello-flake";
  };

  outputs = { self, nixpkgs, flake-utils, hello-flake }:
```

```
 9    flake-utils.lib.eachDefaultSystem (system:
10      let
11        pkgs = import nixpkgs { inherit system; };
12      in
13      {
14        devShells = rec {
15          default = pkgs.mkShell {
16            buildInputs = [ hello-flake.packages.${system}.hello ];
17          };
18        };
19      }
20    );
21 }
```

Line 5 adds `hello-flake` as an input to this flake. Line 8 allows the output function to reference `hello-flake`. Line 16 adds `hello-flake` as a build input for this flake.

Let's take a closer look at the `buildInputs` expression from line 16.

```
hello-flake.packages.${system}.hello
```

Why is the first part `hello-flake` and the last part `hello`? The first part refers to the name we assigned in the input section of our flake, and the last part is the name of the package or app we want. (See Section 4.1, "Flake outputs" for how to identify flake outputs.)

Here's a demonstration using the shell.

```
$ hello-flake     # Fails; dependency not available
bash: line 35: hello-flake: command not found
$ nix develop
$ hello-flake     # Works in development environment
Hello from your flake!
```

10.4.3. Access a Haskell library package in the nixpkgs repo (without using a `.cabal` file)

Occasionally you might want to run a short Haskell program that depends on a Haskell library, but you don't want to bother writing a cabal file. In this example, we will access the `extra` package from the `haskellPackages` set in the nixpkgs repo.

For non-trivial Haskell development projects, it's usually more convenient to use `haskell-flake` as described in Section 9.2.5, "The Nix flake", together with the *high-level workflow* described in [_development_workflows].

flake.nix

```
1 {
```

```
 2   inputs = {
 3     nixpkgs.url = "github:NixOS/nixpkgs";
 4     flake-utils.url = "github:numtide/flake-utils";
 5   };
 6
 7   outputs = { self, nixpkgs, flake-utils }:
 8     flake-utils.lib.eachDefaultSystem (system:
 9       let
10         pkgs = import nixpkgs { inherit system; };
11         customGhc = pkgs.haskellPackages.ghcWithPackages (p: with p; [ p.extra ]);
12       in
13       {
14         devShells = rec {
15           default = pkgs.mkShell {
16               buildInputs = [ customGhc ];
17           };
18         };
19       }
20     );
21 }
```

Line 12 makes a custom GHC that knows about `extra`, and line 16 makes that custom GHC available in the development environment.

Here's a short Haskell program that uses the new flake.

Main.hs

```
1 import Data.List.Extra
2
3 main :: IO ()
4 main = do
5   print $ lower "ABCDE"
6   print $ upper "XYZ"
```

Here's a demonstration using the program.

```
$ runghc Main.hs      # Fails; dependency not available
Main.hs:1:1: error: [GHC-87110]
    Could not find module ‘Data.List.Extra’.
    Use :set -v to see a list of the files searched for.
  |
1 | import Data.List.Extra
  | ^^^^^^^^^^^^^^^^^^^^^^

$ nix develop
$ runghc Main.hs      # Works in development environment
"abcde"
"XYZ"
```

10.4.4. Set an environment variable

Set the value of the environment variable FOO to “bar”.

flake.nix

```
{
  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs";
    flake-utils.url = "github:numtide/flake-utils";
  };

  outputs = { self, nixpkgs, flake-utils }:
    flake-utils.lib.eachDefaultSystem (system:
      let
        pkgs = import nixpkgs { inherit system; };
      in
      {
        devShells = rec {
          default = pkgs.mkShell {
            shellHook = ''
              export FOO="bar"
            '';
          };
        };
      }
    );
}
```

Here’s a demonstration using the shell.

```
$ echo "FOO=${FOO}"
FOO=
$ nix develop
$ echo "FOO=${FOO}"
FOO=bar
```

10.4.5. Access a non-flake package (not in nixpkgs)

In this example, we will use a nix package (not a flake) defined in a remote git repo. However, you can use any of the flake reference styles defined in Section 10.1.2, “Run a flake”.

The `hello-nix` repo provides a `default.nix`. If the derivation in that file allows us to supply our own package set, then our flake can call it to build `hello nix`. If instead it requires `<nixpkgs>`, it is not pure and we cannot use it.

For example, if the file begins with an expression such as

```
with (import <nixpkgs> {});
```

then it requires `nixpkgs` so we cannot use it. Instead, we have to write our own derivation (see Section 10.4.5.2, “If the nix derivation requires `nixpkgs`”).

Fortunately the file begins with

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

then it accepts a package set as an argument, only using `<nixpkgs>` if no argument is provided. We can use it directly to build `hello-nix` (see Section 10.4.5.1, “If the nix derivation does not require nixpkgs”).

If the nix derivation does not require nixpkgs

flake.nix

```
1  {
2    inputs = {
3      nixpkgs.url = "github:NixOS/nixpkgs";
4      flake-utils.url = "github:numtide/flake-utils";
5      hello-nix = {
6        url = "git+https://codeberg.org/mhwombat/hello-nix";
7        flake = false;
8      };
9    };
10
11   outputs = { self, nixpkgs, flake-utils, hello-nix }:
12     flake-utils.lib.eachDefaultSystem (system:
13       let
14         pkgs = import nixpkgs {
15           inherit system;
16         };
17         helloNix = import hello-nix { inherit pkgs; };
18       in
19       {
20         devShells = rec {
21           default = pkgs.mkShell {
22             packages = [ helloNix ];
23           };
24         };
25       }
26     );
27 }
```

Lines 5-8 fetches the git repo for `hello-nix`. However, it is not a flake, so we have to build it; this is done in line 15.

Here's a demonstration using the shell.

```
$ hello-nix     # Fails outside development shell
bash: line 53: hello-nix: command not found
$ nix develop
$ hello-nix
Hello from your nix package!
```

If the nix derivation requires `nixpkgs`

In this case, we need to write the derivation ourselves. We can use `default.nix` (from the hello-nix repo) as a model. Line 15 should be replaced with:

```
      helloNix = pkgs.stdenv.mkDerivation {
        name = "hello-nix";
        src = hello-nix;
        installPhase =
          ''
            mkdir -p $out/bin
            cp $src/hello-nix $out/bin/hello-nix
            chmod +x $out/bin/hello-nix
          '';
      };
```

10.5. Build/runtime environments

10.5.1. Access a top level package from the Nixpkgs/NixOS repo

In Section 9.1.3, "Defining the package", we wrote a shell script that needed access to the Linux `cowsay` command. To accomplish this, we added a step to the `installPhase` that modified that script by including the full path to `cowsay`.

Using `writeShellApplication`

Let's see a different approach. This time we won't start with an existing script; we'll create it during installation. The `writeShellApplication` will modify the script, filling in the shebang and setting up the path with the dependencies specified in `runtimeInputs`. For more information, see the documentation.

flake.nix

```
1 {
2   description = "a very simple and friendly flake";
3
4   inputs = {
5     nixpkgs.url = "github:NixOS/nixpkgs";
6     flake-utils.url = "github:numtide/flake-utils";
```

```
  };

  outputs = { self, nixpkgs, flake-utils }:
    flake-utils.lib.eachDefaultSystem (system:
      let
        pkgs = import nixpkgs { inherit system; };
      in
      {
        devShells = rec {
          default = pkgs.mkShell {
            packages = [ pkgs.cowsay ];
          };
        };

        packages = rec {
          hello = pkgs.writeShellApplication {
            name = "hello-cow";

            runtimeInputs = [
              pkgs.cowsay
            ];

            text = ''
              cowsay "hello";
            '';
          };
          default = hello;
        };

        apps = rec {
          hello = flake-utils.lib.mkApp { drv = self.packages.${system}.hello; };
          default = hello;
        };
      }
    );
}
```

We can build the flake to see how it modifies the script.

```
$ nix build
$ cat result/bin/hello-cow
#!/nix/store/f15k3dpilmiyv6zgpib289rnjykgr1r4-bash-5.3p9/bin/bash
set -o errexit
set -o nounset
set -o pipefail

export PATH="/nix/store/dzi68gvj7y25iidp3lyha6cwzi3rgqwr-cowsay-3.8.4/bin:$PATH"

cowsay "hello";
```

Here's the flake in action.

```
$ nix run
 _______
< hello >
 -------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Other approaches

Nix provides a variety of functions that help with common tasks such as creating scripts. See the section on trivial build helpers in the Nixpkgs reference manual, particularly writeShellScript and writeShellScriptBin.

10.5.2. Access a flake

In this example, we will use a flake defined in a remote git repo. However, you can use any of the flake reference styles defined in Section 10.1.2, "Run a flake".

hello-again

```
1 #!/usr/bin/env sh
2
3 echo "I'm a flake, and I'm running a command defined in a another flake."
4 hello-flake
```

flake.nix

```
 1 {
 2   inputs = {
 3     nixpkgs.url = "github:NixOS/nixpkgs";
 4     flake-utils.url = "github:numtide/flake-utils";
 5     hello-flake.url = "git+https://codeberg.org/mhwombat/hello-flake";
 6   };
 7
 8   outputs = { self, nixpkgs, flake-utils, hello-flake }:
 9     flake-utils.lib.eachDefaultSystem (system:
10       let
11         pkgs = import nixpkgs { inherit system; };
12       in
13       {
14         packages = rec {
15           hello = pkgs.stdenv.mkDerivation rec {
16             name = "hello-again";
17
```

```
18            src = ./.;
19
20            unpackPhase = "true";
21
22            buildPhase = ":";
23
24            installPhase =
25              ''
26                mkdir -p $out/bin
27                cp $src/hello-again $out/bin
28                chmod +x $out/bin/hello-again
29
30                # modify the hello-again script so it can find hello-flake
31                HELLO=$(type -p hello-flake)
32                sed "s_hello-flake_"$HELLO"_" --in-place $out/bin/hello-again
33              '';
34
35            buildInputs = [ hello-flake.packages.${system}.hello ];
36          };
37          default = hello;
38        };
39
40        apps = rec {
41          hello = flake-utils.lib.mkApp { drv = self.packages.${system}.hello; };
42          default = hello;
43        };
44      }
45    );
46 }
```

Line 5 adds `hello-flake` as an input to this flake, Line 8 allows the output function to reference `hello-flake`. As expected, we need to add `hello-flake` as a build input, which we do in line 35. Let's take a closer look at the `buildInputs` expression from line 35.

```
hello-flake.packages.${system}.hello
```

Why is the first part `hello-flake` and the last part `hello`? The first part refers to the name we assigned in the input section of our flake, and the last part is the name of the package or app we want. (See Section 4.1, "Flake outputs" for how to identify flake outputs.)

Line 35 does make `hello-flake` available at build and runtime, but it doesn't put it on the path, so our `hello-again` script won't be able to find it. There are various ways to deal with this problem. In this case we simply edited the script (lines 30-32) as we install it, by specifying the full path to `hello-nix`.

When you're packaging a program written in a more powerful language such as Haskell, Python, Java, C, C#, or Rust, the language build system will usually do all that is required to make the dependencies visible to the program at runtime. In

> that case, adding the dependency to `buildInputs` is sufficient.

Here's a demonstration using the flake.

```
$ nix run
this derivation will be built:
  /nix/store/nham5w86hvivy95k4i5i1cck51qkyxxc-hello-again.drv
building '/nix/store/nham5w86hvivy95k4i5i1cck51qkyxxc-hello-again.drv'...
I'm a flake, and I'm running a command defined in a another flake.
Hello from your flake!
```

10.5.3. Access a Haskell library package in the nixpkgs repo

If you use `haskell-flake` (see Section 9.2.5, "The Nix flake") nothing special needs to be added to `flake.nix`. Simply list your Haskell package dependencies to your cabal file as you normally would.

10.5.4. Access to a Haskell package defined in a flake

In this example we will access three Haskell packages (`pandoc-linear-table`, `pandoc-logic-proof`, and `pandoc-columns`) that are defined as flakes on my hard drive. We are using `haskell-flake`, so the development environment is set up automatically; no need to define `devShells`.

flake.nix

```
{
  description = "Pandoc build system for maths web";

  inputs = {
    nixpkgs.url = "github:nixos/nixpkgs/nixpkgs-unstable";
    flake-parts.url = "github:hercules-ci/flake-parts";
    haskell-flake.url = "github:srid/haskell-flake";
    pandoc-linear-table.url = "/home/amy/github/pandoc-linear-table";
    pandoc-logic-proof.url = "/home/amy/github/pandoc-logic-proof";
    pandoc-columns.url = "/home/amy/github/pandoc-columns";
    pandoc-query.url = "/home/amy/codeberg/pandoc-query";
  };
  outputs = inputs@{ self, nixpkgs, flake-parts, pandoc-linear-table, pandoc-logic-proof, pandoc-columns, pandoc-query, ... }:
    flake-parts.lib.mkFlake { inherit inputs; } {
      systems = nixpkgs.lib.systems.flakeExposed;
      imports = [ inputs.haskell-flake.flakeModule ];

      perSystem = { self', pkgs, ... }: {
        haskellProjects.default = {
      # use my versions of some Haskell pagkages instead of the nixpkgs versions
      packages = {
        pandoc-linear-table.source = inputs.pandoc-linear-table;
        pandoc-logic-proof.source = inputs.pandoc-logic-proof;
        pandoc-columns.source = inputs.pandoc-columns;
```

```
        pandoc-query.source = inputs.pandoc-query;
      };
    };

      # haskell-flake doesn't set the default package, but you can do it here.
      packages.default = self'.packages.pandoc-maths-web;
    };
  };
}
```

10.5.5. Access a non-flake package (not in nixpkgs)

In this example, we will use a nix package (not a flake) defined in a remote git repo. However, you can use any of the flake reference styles defined in Section 10.1.2, “Run a flake”.

We already covered how to add a non-flake input to a flake and build it in Section 10.4.5, “Access a non-flake package (not in nixpkgs)”; here we will focus on making it available at runtime. We will write a flake to package a very simple shell script. All it does is invoke `hello-nix`, which is the input we added earlier.

hello-again

```
#!/usr/bin/env sh

echo "I'm a flake, but I'm running a command defined in a non-flake package."
hello-nix
```

flake.nix

```
{
  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs";
    flake-utils.url = "github:numtide/flake-utils";
    hello-nix = {
      url = "git+https://codeberg.org/mhwombat/hello-nix";
      flake = false;
    };
  };

  outputs = { self, nixpkgs, flake-utils, hello-nix }:
    flake-utils.lib.eachDefaultSystem (system:
      let
        pkgs = import nixpkgs { inherit system; };
        helloNix = import hello-nix { inherit pkgs; };
      in
      {
        packages = rec {
          hello = pkgs.stdenv.mkDerivation rec {
            name = "hello-again";
```

```
21
22            src = ./.;
23
24            unpackPhase = "true";
25
26            buildPhase = ":";
27
28            installPhase =
29              ''
30                mkdir -p $out/bin
31                cp $src/hello-again $out/bin
32                chmod +x $out/bin/hello-again
33
34                # modify the hello-again script so it can find hello-nix
35                HELLO=$(type -p hello-nix)
36                sed "s_hello-nix_"$HELLO"_" --in-place $out/bin/hello-again
37              '';
38
39
40            buildInputs = [ helloNix ];
41          };
42          default = hello;
43        };
44
45        apps = rec {
46          hello = flake-utils.lib.mkApp { drv = self.packages.${system}.hello; };
47          default = hello;
48        };
49      }
50    );
51 }
```

Lines 5-8 and 15 were explained in Section 10.4.5, “Access a non-flake package (not in nixpkgs)”. As expected, we need to add `helloNix` as a build input, which we do in line 40. That does make it available at build and runtime, but it doesn’t put it on the path, so our `hello-again` script won’t be able to find it.

There are various ways to deal with this problem. In this case we simply edited the script (lines 34-36) as we install it, by specifying the full path to `hello-nix`.

When you’re packaging a program written in a more powerful language such as Haskell, Python, Java, C, C#, or Rust, the language build system will usually do all that is required to make the dependencies visible to the program at runtime. In that case, adding the dependency to `buildInputs` is sufficient.

Here’s a demonstration using the flake.

```
$ nix run
this derivation will be built:
```

```
  /nix/store/ik73sxc11b8q52y8m8b6k5bppqfym99l-hello-again.drv
building '/nix/store/ik73sxc11b8q52y8m8b6k5bppqfym99l-hello-again.drv'...
I'm a flake, but I'm running a command defined in a non-flake package.
Hello from your nix package!
```

10.6. An (old-style) Nix shell with access to a flake

If you are maintaining legacy code, you may need to provide access to a flake in a `nix-shell`. Here's how.

shell.nix

```
with (import <nixpkgs> {});
let
   hello-flake = ( builtins.getFlake
                       git+https://codeberg.org/mhwombat/hello-
  flake?ref=main&rev=3aa43dbe7be878dde7b2bdcbe992fe1705da3150
         ).packages.${builtins.currentSystem}.default;
in
mkShell {
  buildInputs = [
    hello-flake
  ];
}
```

Here's a demonstration using the shell.

```
$ nix-shell
copying path '/nix/store/9zkfbyk0xjiyzkky8p01jc4hm7jrwz5w-source' from
'https://cache.nixos.org'...
this derivation will be built:
  /nix/store/qspvnv4m9cz2rinvswagi8jfnclzx24k-hello-flake-20230218.drv
these 36 paths will be fetched (65.28 MiB download, 296.69 MiB unpacked):
  /nix/store/218mzh306bw7rkhcgljlwqpvrdmcy13i-acl-2.3.1
  /nix/store/ifs8pac5sv026ynk5gk5qg6ap7qdq1x1-attr-2.5.1
  /nix/store/zcla0ljiwpg5w8pvfagfjq1y2vasfix5-bash-5.1-p16
  /nix/store/c4j39cgj3cwxgb5fp3z1vsyjhav33bxs-binutils-2.39
  /nix/store/4c302k97bbc33xgjzmkypkwrmqba9gqy-binutils-wrapper-2.39
  /nix/store/61rpfcaxhyqfmnk5qp4z7hf20wh9zgrk-bzip2-1.0.8
  /nix/store/7yhklbxbjivcqk5ql4rl660glgc3iba9-bzip2-1.0.8-bin
  /nix/store/gn5515zj8skk23jvrrljirgxr10fw9az-coreutils-9.1
  /nix/store/wjiawwzad72q2mm9515fpx6n4zn91xfg-diffutils-3.8
  /nix/store/x056x7nsrzfxpzf61pfc6apajax7zd4h-ed-1.18
  /nix/store/kny46nl4gq9415qqnhmwpfvn2376399m-expand-response-params
  /nix/store/pdds64xdjvzddhcxlidna5xyphmjdxpg-file-5.43
  /nix/store/3iwxvsnfx6kr7ga37kbmjz1df3nb95ri-findutils-4.9.0
  /nix/store/n5x3p2n835bl0v6ampnqchdklwvr8szh-gawk-5.1.1
  /nix/store/3cjvw93ly6cx2af13f2l3pw4yzbi8wp6-gcc-11.3.0
  /nix/store/b13h86pg7lbf6vpc1vwzw6akmakyw1bs-gcc-11.3.0-lib
```

```
  /nix/store/9yxrwp848f7msm6m2442yfpji8m3206b-gcc-wrapper-11.3.0
  /nix/store/9xfad3b5z4y00mzmk2wnn4900q0qmxns-glibc-2.35-224
  /nix/store/gfsg0n19r8q8v69kwwjvx4m2y77vi7i8-glibc-2.35-224-bin
  /nix/store/bq928ff6m7lvcfyvcdvgvqxhqi5f3ijq-glibc-2.35-224-dev
  /nix/store/askxicd00lc1jqshyh5lcwjpxfqygiiy-gmp-with-cxx-stage4-6.2.1
  /nix/store/6l34p4ip6ib1q87cpd4g5yhg3j86zign-gnugrep-3.7
  /nix/store/qnhmjdxh35whjqqya1sd6jh7jh2ld978-gnumake-4.3
  /nix/store/687xxx92x0pg1yzvj8lv4xz5b9vs20qh-gnused-4.8
  /nix/store/nybc3q7ksr0i226cvs7wgrfgzx3b60nw-gnutar-1.34
  /nix/store/m8gycf587qfdkjc76bsf6c6w12pbzc35-gzip-1.12
  /nix/store/5mh5019jigj0k14rdnjam1xwk5avn1id-libidn2-2.3.2
  /nix/store/34xlpp3j3vy7ksn09zh44f1c04w77khf-libunistring-1.0
  /nix/store/i38jcxrwa4fxk2b7acxircpi399kyixw-linux-headers-6.0
  /nix/store/c4ssk8c0y9hxwjgv9i1qh26sas9as0hg-patch-2.7.6
  /nix/store/d84xrx5n73fb0bc28gnsgwvj1rv2fywk-patchelf-0.15.0
  /nix/store/sm5ch5nkhpj4dmhm8siia1kq8sia6jf5-pcre-8.45
  /nix/store/7bc3vsg3laghwg0cin2il9iwd7ka820k-stdenv-linux
  /nix/store/2gsz9a3v0fylbghzsljz2dnxfs9iidrn-xz-5.2.7
  /nix/store/kvqhx6wxflyp780ggpd27dklwa28sban-xz-5.2.7-bin
  /nix/store/fblaj5ywkgphzpp5kx41av32kls9256y-zlib-1.2.13
copying path '/nix/store/34xlpp3j3vy7ksn09zh44f1c04w77khf-libunistring-1.0' from
'https://cache.nixos.org'...
copying path '/nix/store/i38jcxrwa4fxk2b7acxircpi399kyixw-linux-headers-6.0' from
'https://cache.nixos.org'...
copying path '/nix/store/5mh5019jigj0k14rdnjam1xwk5avn1id-libidn2-2.3.2' from
'https://cache.nixos.org'...
copying path '/nix/store/9xfad3b5z4y00mzmk2wnn4900q0qmxns-glibc-2.35-224' from
'https://cache.nixos.org'...
copying path '/nix/store/ifs8pac5sv026ynk5gk5qg6ap7qdq1x1-attr-2.5.1' from
'https://cache.nixos.org'...
copying path '/nix/store/zcla0ljiwpg5w8pvfagfjq1y2vasfix5-bash-5.1-p16' from
'https://cache.nixos.org'...
copying path '/nix/store/b13h86pg7lbf6vpc1vwzw6akmakyw1bs-gcc-11.3.0-lib' from
'https://cache.nixos.org'...
copying path '/nix/store/x056x7nsrzfxpzf61pfc6apajax7zd4h-ed-1.18' from
'https://cache.nixos.org'...
copying path '/nix/store/687xxx92x0pg1yzvj8lv4xz5b9vs20qh-gnused-4.8' from
'https://cache.nixos.org'...
copying path '/nix/store/kny46nl4gq9415qqnhmwpfvn2376399m-expand-response-params' from
'https://cache.nixos.org'...
copying path '/nix/store/n5x3p2n835bl0v6ampnqchdklwvr8szh-gawk-5.1.1' from
'https://cache.nixos.org'...
copying path '/nix/store/gfsg0n19r8q8v69kwwjvx4m2y77vi7i8-glibc-2.35-224-bin' from
'https://cache.nixos.org'...
copying path '/nix/store/qnhmjdxh35whjqqya1sd6jh7jh2ld978-gnumake-4.3' from
'https://cache.nixos.org'...
copying path '/nix/store/61rpfcaxhyqfmnk5qp4z7hf20wh9zgrk-bzip2-1.0.8' from
'https://cache.nixos.org'...
copying path '/nix/store/m8gycf587qfdkjc76bsf6c6w12pbzc35-gzip-1.12' from
'https://cache.nixos.org'...
copying path '/nix/store/sm5ch5nkhpj4dmhm8siia1kq8sia6jf5-pcre-8.45' from
```

```
'https://cache.nixos.org'...
copying path '/nix/store/2gsz9a3v0fylbghzsljz2dnxfs9iidrn-xz-5.2.7' from
'https://cache.nixos.org'...
copying path '/nix/store/fblaj5ywkgphzpp5kx41av32kls9256y-zlib-1.2.13' from
'https://cache.nixos.org'...
copying path '/nix/store/218mzh306bw7rkhcgljlwqpvrdmcy13i-acl-2.3.1' from
'https://cache.nixos.org'...
copying path '/nix/store/7yhklbxbjivcqk5ql4rl660glgc3iba9-bzip2-1.0.8-bin' from
'https://cache.nixos.org'...
copying path '/nix/store/pdds64xdjvzddhcxlidna5xyphmjdxpg-file-5.43' from
'https://cache.nixos.org'...
copying path '/nix/store/c4ssk8c0y9hxwjgv9i1qh26sas9as0hg-patch-2.7.6' from
'https://cache.nixos.org'...
copying path '/nix/store/nybc3q7ksr0i226cvs7wgrfgzx3b60nw-gnutar-1.34' from
'https://cache.nixos.org'...
copying path '/nix/store/6l34p4ip6ib1q87cpd4g5yhg3j86zign-gnugrep-3.7' from
'https://cache.nixos.org'...
copying path '/nix/store/kvqhx6wxflyp780ggpd27dklwa28sban-xz-5.2.7-bin' from
'https://cache.nixos.org'...
copying path '/nix/store/bq928ff6m7lvcfyvcdvgvqxhqi5f3ijq-glibc-2.35-224-dev' from
'https://cache.nixos.org'...
copying path '/nix/store/c4j39cgj3cwxgb5fp3z1vsyjhav33bxs-binutils-2.39' from
'https://cache.nixos.org'...
copying path '/nix/store/askxicd00lc1jqshyh5lcwjpxfqygiiy-gmp-with-cxx-stage4-6.2.1'
from 'https://cache.nixos.org'...
copying path '/nix/store/d84xrx5n73fb0bc28gnsgwvj1rv2fywk-patchelf-0.15.0' from
'https://cache.nixos.org'...
copying path '/nix/store/3cjvw93ly6cx2af13f2l3pw4yzbi8wp6-gcc-11.3.0' from
'https://cache.nixos.org'...
copying path '/nix/store/gn5515zj8skk23jvrrljirgxr10fw9az-coreutils-9.1' from
'https://cache.nixos.org'...
copying path '/nix/store/wjiawwzad72q2mm9515fpx6n4zn91xfg-diffutils-3.8' from
'https://cache.nixos.org'...
copying path '/nix/store/3iwxvsnfx6kr7ga37kbmjz1df3nb95ri-findutils-4.9.0' from
'https://cache.nixos.org'...
copying path '/nix/store/4c302k97bbc33xgjzmkypkwrmqba9gqy-binutils-wrapper-2.39' from
'https://cache.nixos.org'...
copying path '/nix/store/9yxrwp848f7msm6m2442yfpji8m3206b-gcc-wrapper-11.3.0' from
'https://cache.nixos.org'...
copying path '/nix/store/7bc3vsg3laghwg0cin2il9iwd7ka820k-stdenv-linux' from
'https://cache.nixos.org'...
building '/nix/store/qspvnv4m9cz2rinvswagi8jfnclzx24k-hello-flake-20230218.drv'...
unpacking sources
patching sources
configuring
no configure script, doing nothing
building
installing
post-installation fixup
shrinking RPATHs of ELF executables and libraries in
/nix/store/h1064a2fs25vx6zwr0fmfrhfxhw6qm7z-hello-flake-20230218
```

```
strip is /nix/store/9yxrwp848f7msm6m2442yfpji8m3206b-gcc-wrapper-11.3.0/bin/strip
stripping (with command strip and flags -S) in
/nix/store/h1064a2fs25vx6zwr0fmfrhfxhw6qm7z-hello-flake-20230218/bin
patching script interpreter paths in /nix/store/h1064a2fs25vx6zwr0fmfrhfxhw6qm7z-
hello-flake-20230218
/nix/store/h1064a2fs25vx6zwr0fmfrhfxhw6qm7z-hello-flake-20230218/bin/hello-flake:
interpreter directive changed from "#!/usr/bin/env sh" to
"/nix/store/zcla0ljiwpg5w8pvfagfjq1y2vasfix5-bash-5.1-p16/bin/sh"
checking for references to /build/ in /nix/store/h1064a2fs25vx6zwr0fmfrhfxhw6qm7z-
hello-flake-20230218...
$ hello-flake
Hello from your flake!
```