The Nix Package Manager

by Jonathan Ringer and Tim Deherra

This book assumes usage of the nix 3.0 cli. Which can be enabled in nix 2.4+ following these instructions.

This book is available on the web at https://book.divnix.com/. For changes to the book, please see the https://github.com/divnix/nix-book.

Preface

My journey to learn nix was only possible by my extreme desire to master it. The path was anything but easy and predictable. And it is still a considerable hurdle for many trying to learn nix.

The goal of this book is to provide newcomers with a more approachable document than the nix-pills series. Although nix-pills is still a very good resource with many years of refinement, it is extensive and hard to follow without some prior knowledge of nix. This book hopes to provide a more recent account of nix with more of a focus on giving the user intuition around what nix is doing rather than a deep understanding like nix-pills.

The goal is not to replace any existing nix guides or documentation, but rather provide a good starting place for new users. Motivation for writing this is to provide a "nix equilavent of the rust-lang book". Where there is one resource which can be read end-to-end in an afternoon and able to equip the reader with the knowledge necessary to thrive in the nix ecosystem.

-- Jonathan Ringer

Introduction

The Nix Package Manager

Nix is a package manager which focuses on caputuring all inputs which contribute to building software. The result of factoring all of the information about building the software is called a derivation. This aggregated information includes where the source code is pulled, configuration flags, patches, dependencies, build steps, installation steps, and many other

potential inputs.

This information is aggregated through hashing, and allows nix to describe and reference the exact software which is intended to use. This enables nix to be used on any system because it's assumptions do not collide with the assumptions of a host system. This also means that nix does not adhere to the traditional File Hierarchical System(FHS) but it also means that it's not limited to FHS's restriction of only having a single variant of a piece of software.

Who is Nix For

Teams of Developers

Development needs to have similar tooling across every individual. Having divergent development environments and productions environments is a major cause of regressions in software development. Nix can help mitigate this by allowing development environments to be version controlled and maintained along with a project.

DevOps (Operations)

Nix allows you to percisely describe the software you intended to use. Nix packages are defined by their dependencies, so they inherently retain their SBOM (Software Bill of Materials) by default. By leveraging NixOS modules, one can also create configurable services and compose it into coherent systems. The combination of Nix + NixOS allows you to have delcarative configuration of both services and sytems.

System Administrators (home to enterprise)

Ever have to maintain a few systems to a few hundred systems? The ability to version control and manipulate systems-as-code enables a new paradigm of system configuration management. Atomically apply or rollback system updates for each system. Nixpkgs can also be freely extended to include personal or private additions of software; this allows you to leverage all other Nix tooling as though your application-specific software was a first-class citizen.

Also, Nix largely invalidates the need for docker. However, nix can also be used to produce docker images if there is a downstream technology which consumes oci images as an interface (E.g. kubernetes).

Power Users

Do you have incredible specific or opinated environments? Nix allows you to declaratively create project (flakes), user (home-manager), or system (NixOS) environments with the exact software and configuration you desired. Whether you're building software or ricing a desktop, nix will allow you to version control and specify your configuration exactly how you intended.

The Nix Ecosystem

There's roughly four layers of abstractions in the official nix ecosystem, these are:

- Nix - The domain-specifc language used to write nix expressions
- Nix - The package manager
- Nixpkgs - The official Nix package repository
- NixOS - A linux distribution built upon nixpkgs

There are also a few unofficial projects which are commonly used within the community:

- Home-manager - NixOS-like user configuration for linux or MacOS built upon nixpkgs
- Nix-darwin - NixOS-like configuration, but for MacOS

All of these topics will be discussed in greater detail in later sections, but a quick summary of official projects are provided below.

The Nix Language

The Nix language is a Domain-Specific Language (DSL) which is designed to handle package configuration. Nix can be thought of JSON + functions + some syntax sugar. It's main goal is to provide effect-free evaluation of package configuration, to this point Nix is restricted in many ways and lacks many features from generic programming languages. There is very limited input and output possible to the system, there are no loops, no concurrency primitives, and no types. What is left is a small functional-oriented programming language.

After all, Nix's goal is to take a few inputs such as a system platform, and produce a build graph which can be used to build software.

Nix the Package Manager

The Nix Package Manager began its life as the PhD thesis work of Eelco Dolstra. The goal was to bring discipline to the software landscape. Similar to how structured programming helped tame the complexity of goto through introducing constructs such as loops and logic flow; so too does nix attempt to tame the chaos of package management through explicit descriptions of software and their dependencies. The truely novel idea of Nix is that of the *derivation*. It encapsulates everything about a piece of software, and these derivations can be referenced from other derivations constituting a Directed-Acyclic-Graph of how to built that software from source.

Nixpkgs

Nixpkgs is the official package repository for the Nix community. It contains the logic on how to build over 60,000+ software packages. Nixpkgs can be thought of as an expert body-of-knowledge on the subject of how to build software. When a user asks for the "firefox" package, the nix package manager is able to input the user's computer platform into nixpkgs, and nixpkgs is then able to produce a build graph on how to build firefox and all of it's dependencies down to the C compiler. This allows for a great deal of freedom in how nix is leveraged, and nix can be used on any Linux distribution and MacOS as first class supported OS's, and to a much lesser degree on many other UNIX-like OS's.

Nixpkgs is also supported by Hydra, which provides pre-built binaries of libre software for Linux and MacOS.

NixOS

NixOS is a non-FHS Linux distribution which leverages nixpkgs to provide a wealth of software ready to be combined into a system environment. The concept of a nix derivation is extended here to include service configuration and system creation. The entirity of the system is represented as a derivation which gives it many of it's defining qualities such as atomic rollbacks, system-as-a-configuration-file, extensive user configuration potential.

Installation

Linux and MacOS

The guided installer is the preferred way to install nix, please run the following in a shell:

```
sh <(curl -L https://nixos.org/nix/install) --daemon
```

Other Installation Methods

There are many ways to leverage nix, for more installation options, please visit the official download page.

Derivations

Derivations are the defining feature of nix. Derivations attempt to capture everything that it would take to build a package. This includes, but is not limited to: source code, dependencies, build flags, build and installation steps, tests, and environment variables. The culmination of all direct and transitive build depdencies is commonly refered to as the derivations's "build closure". More dependencies that a package refers to, more package will need to be created in order to attempt a build. Generally, dependencies of a derivation are other derivations.

Types of Derivations

Fixed Output Derivations (FODs)

These are the "leaves" of any build closure, in that, they do not refer to other derivations. These derivations are defined by their content. These derivations are easily differientiated because they will contain a sha256 (or other hash) which is used to enforce that an artifact is reproducible.

One critical difference from evaluated derivations is that Fixed-Output derivations are able

to have access to the network while fetching contents. This "impurity" is offset by enforcing that the hash matches, and reproducibility is delegated to the process which fetchs the assets.

Many of the `fetch*` utilities in nixpkgs and nix's builtins will create FODs.

Input-Addressed Derivations

Input-Addressed derivations are generally what are referred to when the term derivation is used. These derivations are defined by all of the dependencies, build phases, and flags present during a build. Nix captures all of the variables which constitute a derivation and uses a cryptographic hash to give each derivation a unique name.

`stdenv.mkDerivation` and related `build*` helpers will create an input-addressed derivation.

Content-Addressable Derivations (CA Derivations)

NOTE: CA Derivations are still considered experimental at the time of writing

Content-Addressable (CA) derivations are a hybrid of both FOD and IA derivations. The problem which CA derivations address are rebuilds. In the IA derivation model, a patch to openssl will cause all downstream packages to rebuild since that derivation will propagate the patch change across all consumers. Under CA derivations, nix can determine that a consuming package which was built before the openssl patch has remained unchanged with the only exception being where openssl is located in the nix store. In this case the package which uses openssl is "the same" in usage, the only thing which has changed is what variant of openssl it uses. Nix is then free to assert an equivalence of the package before and after the openssl patch; thus, it doesn't need to rebuild all packages, just update the references of openssl.

The name Content-Addressable comes from the fact that the implementation will stub out nix store paths and use this normalized content to compare against other builds. Now nix can deduplicate builds which were done previously. In the openssl example, the build of curl will likely be exactly the same; thus any package which just consumes curl will not have to be rebuilt. Only the references to the new variant of curl needs to be updated.

CA derivations are an opt-in experimental feature, but don't require the user to alter their existing workflows.

Create a Derivation

Before a package is built, a derivation must be created. The derivation can be thought of as the unamiguous definition of how to build a package. The process of creating a derivation is called "instantiatiation", or sometimes also refered to as evaluation (although this is more general). Every package in nixpkgs has a corresponding derivation. This means that we can create and inspect the derivation for anything exposed in nixpkgs. An example would be:

```
$ nix-instantiate '<nixpkgs>' -A hello
/nix/store/byqskk0549v1zz1b2a61lb7llfn4h5bw-hello-2.10.drv

# or using flakes, nix>=2.4

$ nix eval nixpkgs#hello.drvPath
"/nix/store/byqskk0549v1zz1b2a61lb7llfn4h5bw-hello-2.10.drv"
```

Inspect the contents of a derivation

To inspect the contents of the drv, one can use the `nix show-derivation` utility.

```
$ nix show-derivation /nix/store/byqskk0549v1zz1b2a61lb7llfn4h5bw-hello-2.10.drv
{
  "/nix/store/byqskk0549v1zz1b2a61lb7llfn4h5bw-hello-2.10.drv": {
    "outputs": {
      "out": {
        "path": "/nix/store/f4bywv8hjwl0ckv7l077pnap81h6qxw4-hello-2.10"
      }
...
```

Defining characteristics of a derivation

There's a few important features of a derivation:

- It's a description of how to build the package from source
- The output paths are determined before the build begins
- All dependencies are resolved as part of instantiation, and may have a similar derivation description of their builds
- Any additional flags (makeFlags, configuration flags, cflags, or ld flags) are explicitly stated

- There's no ambiguity. The system, architecture, and other options have been resolved.
- It's immutable. If you want to change a derivation, you need to evaluate a new one.
- It's unique. The hashing scheme ensures that there should only ever be one derivation; if two derivations match, then they are exactly the same in every way.

Realise a derivation

Building a derivation is referred to as "realisation". A derivation is just an abstract description of a package, based upon what it requires to build. Derivations can be thought of as constructing a blueprint, but realisation is the construction of the desired object. Taking from the previous example, one can build a derivations like so:

```
$ nix-store --realise /nix/store/byqskk0549v1zz1b2a61lb7llfn4h5bw-hello-2.10.drv
...
/nix/store/f4bywv8hjwl0ckv7l077pnap81h6qxw4-hello-2.10

# or in nix flakes:

$ nix build nixpkgs#hello
...
/nix/store/f4bywv8hjwl0ckv7l077pnap81h6qxw4-hello-2.10
```

Here, the gnu hello project was built and installed at the output path. This includes: the executable binary, documenation, and locale info.

```
$ tree -L 2 /nix/store/f4bywv8hjwl0ckv7l077pnap81h6qxw4-hello-2.10
/nix/store/f4bywv8hjwl0ckv7l077pnap81h6qxw4-hello-2.10
├── bin
│   └── hello
└── share
    ├── info
    ├── locale
    └── man
```

The `nix-build` and `nix build` commands will perform both instantiation and realisation. These are the most common commands used when iterating on packages. One could also do:

```
$ nix-buid '<nixpkgs>' -A hello
# these are the same, nix build is just much more concise
$ nix-store --realise $(nix-instantiate '<nixpkgs>' -A hello)
```

Note: Many other commands also will realise a derivation as part of a workflow. Some examples are: `nix-shell`, `nix shell`, `nix-env`, `nix run`, and `nix profile`. These commands are very goal oriented and will differ significantly in how they leverage nix, often

realisation is an side-effect to achieve that goal.

Using Derivations from Nix

The derivation is the main abstraction of nix. All of Nixpkgs and NixOS is created by leveraging derivations to create new derivations, scripts, services, and even entire linux distributions. The ability to compose these usecases with uniquely named packages allows nix the freedom to aggressively share common dependencies, meanwhile allowing the flexibility to have potentially incompatible packages available on the system.

The nix language allows for consumption of derivations to be quite transparent. For example:

```
$ cat hello.nix
let
  pkgs = import <nixpkgs> { };
in
  pkgs.writeScriptBin "greet.sh" ''
    ${pkgs.hello}/bin/hello -g "Hello $USER!"
  ''

$ nix-build hello.nix
this derivation will be built:
  /nix/store/xd9qpwnvybm9p8k2szhkvpd2ym85is9p-greet.sh.drv
building '/nix/store/xd9qpwnvybm9p8k2szhkvpd2ym85is9p-greet.sh.drv'...
/nix/store/h8yxaciazc8basn9l335bmdrpfak0aqk-greet.sh

$ cat ./result/bin/greet.sh
/nix/store/mg35qkhk7wqbhhykpakds4fsm1riy8ga-hello-2.12.1/bin/hello -g "Hello
$USER!"

$ ./result/bin/greet.sh
Hello jon!
```

We created a `greet.sh` script which will greet the user. Nix first created the "derivation" (build plan) of our script at `/nix/store/<hash>-greet.sh.drv`, and then realised (built) the derivation as `/nix/store/<hash>-greet.sh`. We can see from the contents of the resulting file that `pkgs.hello` was substituted for the realised output path. This allows for us to not worry about what the unique name of the derivation will be, but rather worry about the contents post realisation.

Although this may not seem markedly better than other package management workflows such as: please install these tools, then run this script. There is quite a lot of benefit to leveraging nix whether it's to create scripts or build more software:

- Use of exact versions which you control
    - For example, which version of python or node do you have?
- No longer dependent on the state of the consuming system
    - For example, do you have python installed?
- Use of multiple versions of the same software
    - Want to use NodeJS v14 in one script, but NodeJS v16 in another? No problem.

Although many ecosystems will have ecosystem specific solutions to these solutions (e.g. tox for python, nvm for node), nix provides a universal abstraction for native dependencies and any downstream dependencies.

Use of "outPath" as a toString

This is one of the oddities of nix, but stringification of an object which contains a key "outPath" will return the contents of the "outPath" key. Since all derivations will have an outPath, any usage of them in a string will yeild the store path that they create.

```
nix-repl> a = { outPath = "foo"; }

nix-repl> "${a} bar"
"foo bar"
```

The Nix Language

Nix is a pure, lazy, functional language which serves a domain-specific language (DSL) for writing nix derivations and expressions. In general, nix can be thought of as JSON with functions.

The goal of Nix is to facilitate the creation of a derivation. In most situations nix is given a small amount input and expected to produce a result (usually a derivation). In the case of nixpkgs, the workflow is generally, "given the user has an x86_64-linux device and the information held within nixpkgs then the desired package will be `/nix/store/<some hash>-<package>`. This is also why the word evaluation is used commonly when refering to nix packages, the nix expression which describes how to build software can evaluate to it's final reduced state given just a system platform.

Nix Language Basics

Primitive Values

These values are mostly similar to JSON:

Type	Description	Example
integer	Whole number	1
float	Floating point number	1.054
string	UTF-8 string	"hello!"
path	File or url	./default.nix

NOTE: Paths are special. They will be resolved relative to the file. The value must contain a "/" to be considered a path, however, it's common to construct the value starting with "." to avoid confusion (e.g. ./foo/bar vs foo/bar). If a path is referenced as part of a package, that path will be added to the nix store, and all references to that path will be substituted with the nix store path.

Strings

Nix exposes two ways to express strings. Strings are enclosed with double quotes: `"hello"`. This works well for small strings, such as simple flags. However, it's common to write a block of commands which need to be executed; for this, nix also has multi-line support with the "lines" construct. "lines" are denoted by two single quotes.

Example usage of lines:

```
postPatch = ''
  ./autogen.sh
  mkdir build
  cd build
'';
```

Here, we have postPatch being assigned a series of commands to be ran as part of a build.

Another quality of lines, is that all shared leading whitespace will be truncated. This allows for the the lines blocks to be adjusted to the indention of the parent nix expression without influencing the contents of the string.

```
$ cat lines.nix
''
  2 spaces
   3 spaces
    4 spaces
''
$ nix eval -f lines.nix
"2 spaces\n 3 spaces\n  4 spaces\n"
```

Lists

Lists work similarly to most other languages, but are whitespace delimited. `[ 1 2 ]` is an array with elements `1` and `2`.

Note: For oddities around lists and elements which use whitespace, please see list common mistakes.

Attribute Set (Attr set)

This can be thought of as a dictionary or map in most other languages. The important distinct is that the keys are always ordered, so that the order doesn't influence how a derivation will produce a hash. Attr sets values do not need to be of the same type. Attr sets are constructed using an `=` sign which denotes key value pairs which are separated with semicolons `;`, the attr set is enclosed with curly braces `{ }`. Selection of an attribute is done through dot-notation `<set>.<key>`.

```
nix-repl> a = { foo = "bar"; count = 5; flags = ''-g -O3''; }
nix-repl> a.count
5

# Shorthand for nested attribute sets
nix-repl> :p { foo.bar.baz = 1; foo.bar.buzz = 2; }
{ foo = { bar = { baz = 1; buzz = 2; }; }; }
```

You will commonly see empty attr sets in nixpkgs, an example being:

```
  hello = callPackage ../applications/misc/hello { };
```

Derivations

Technically, a derivation is just an attr set which has a few special attributes set to valid values which then nix can later realise into a build. Promotion from an attr set to derivation is facilitated through the `builtins.derivation` function. However directly calling the builtin is highly discouraged within nixpkgs. Instead people are encouraged to use stdenv.mkDerivation and other established builders which provide many good defaults to achieve their packaging goals.

If / Else logic

Like many other functional programming languages, you cannot use `if` without an accompanying `else` clause. This is because the expression needs to return a value, not just follow a code path.

```
extension = if stdenv.isDarwin then
    ".dylib"
  else
    ".so";
```

Note: The proper way to find the shared library extension within nixpkgs is `hostPlatform.extensions.sharedLibrary` .

Let expressions

Let expressions are a way to define values to be used later in a given 'in' scope. Generally these are used to alter a given value to conform to a slightly different format. Let expressions can refer to other values defined in the same let scope. For haskell users, let expressions work similarly to how they work in Haskell.

```
src = let
  # e.g. 3.1-2 -> 3_1_2
  srcVersion = lib.strings.replaceStrings [ "." "-" ] [ "_" "_"] version;
  srcUrl = "https://example.com/download/${pname}-${srcVersion}.tar.gz";
in fetchurl {
  url = srcUrl;
  sha256 = "...";
};
```

With expressions

With expressions allows for many values on an attr set to be exposed by their key names.

```
# before
meta = {
  licenses = lib.licenses.cc0;
  maintainers = [ lib.maintainers.jane lib.maintainers.joe ];
  platforms = lib.platforms.unix;
};


# after
meta = with lib; {
  licenses = licenses.cc0;
  maintainers = with maintainers; [ jane joe ];
  platforms = platforms.unix;
};
```

Laziness

Many pure functional programming languages also have the feature that the evaluation model of the language is lazy. This means that the values of a data structure aren't computed until needed. The benefits for nix is that evaluating a package doesn't mean computing all packages, but only computing the dependency graph for the packages requested. In practice this means limiting the scope of an action from 80,000+ possible dependencies to just the dependencies explicitly mentioned by the nix expressions.

Although laziness isn't a hard requirement for nix to work. The purity model of nix makes laziness more a symptom rather than an explicit design goal. However, It does enable many implicit benefits such as memoization.

Functions

Nix only has unary functions: unary functions are functions which only accept one parameter. However, in combination with uncurrying, you can create functions which take an arbitrary number of parameters.

Functions can be treated as values, and freely passed to other functions as such. To name a function, it just needs to be assigned to variable, much as you would do to a literal.

Function examples:

```
# creation, and immediate application of a nameless function
nix-repl> (x: x + 2) 3
5

# assigning a function to a variable, then later applying it
nix-repl> addTwo = x: x + 2

nix-repl> addTwo 3
5

# two parameters
nix-repl> sumBoth = x: y: x + y

nix-repl> sumBoth 2 5
7
```

Attr sets as inputs

Nix also heavily uses attr sets to pass around many arguments. In nixpkgs, this is most commonly used to express what subset of packages and utilities should be used for a nix expression. It's also useful when a large context for a function is needed, and an ordered list of parameters is a poor fit.

Attr sets as inputs are also particular good when the function can provide good defaults, and only a small subset of inputs are expected to be edited.

Function examples:

```
# function which takes an attr set
nix-repl> addTwo = { x }: x + 2

nix-repl> addTwo { x = 3; }
5

# function which takes optional attr set values
nix-repl> addTwoOptional = { x ? 4 }: x + 2

nix-repl> addTwoOptional { }
6

nix-repl> addTwoOptional { x = 5; }
7

# same as above, but binding the entire attr set to another variable
nix-repl> addTwoOptional = { x ? 4 }@args: args.x + 2

nix-repl> addTwoOptional  { x = 6; }
8
```

Note: The `@` syntax is not very common for most nix expressions. Its most common use case are "helpers", which only care about a subset of arguments, and will then call another function with some of the inputs pruned. A good example of this is the `pkgs.fetchFromGithub` fetcher; which will know how to translate owner, repo, rev, and other options into a call to `builtsin.fetchzip` or `builtins.fetchgit`.

Imports and callPackage

Import

`import` is one of the few keywords in nix. It allows for a file to be read and evaluated. If a directory is passed to `import`, then it will assume `<directory>/default.nix` was the desired file.

```
$ cat data.nix
{ a = "foo"; b = "bar"; }

nix-repl> :p import ./data.nix
{ a = "foo"; b = "bar"; }

$ cat expression.nix
5+2

nix-repl> :p import ./expression.nix
7
```

This still extends to functions:

```
$ cat function.nix
{ x, y }: x + y

nix-repl> :p import ./function.nix { x = 2; y = 9; }
11
```

Imports for packages

In nixpkgs, each package usually has a corresponding file associated with the packaging and related concerns of just that package. Early in nix's history, `import` was used to integrate the files with other expressions and allow for greater organization of code. However, the import model is quite explicit, and requires users to declare the dependencies twice.

Below is an example expression for `openssl`:

```
# pkgs/libraries/openssl/default.nix
{ lib, stdenv, fetchurl, perl }:

stdenv.mkDerivation {
  ...
}
```

In the `import` paradigm, the calling site would look:

```
  openssl = import ../libraries/openssl {
    inherit lib stdenv fetchurl perl;
  };
```

Obviously this isn't ideal. The dependencies need to be referred to three times, once at the call site, as inputs to the expression, and then within the expression at the appropriate section. The tediousness of passing the values will be solved by `callPackage`.

CallPackage

`callPackage` is a function which will call a function with the appropriate dependencies. The package set will generally expose a `callPackage` function with the current package set already bound.

A minimal callPackage implementation can be thought of as:

```
  # <nixpkgs>/lib/customisation.nix
  # callPackageWith :: Attr Set -> (Attr Set -> drv) -> Attr Set -> drv
  callPackageWith = autoArgs: fn: args:
  # autoArgs - Attr set of "defaults", for nixpkgs this would be all top-level
packages
  # fn       - A nix expression which uses an attr set as in input.
  # args     - Overrides to the defaults in autoArgs
  let
    # if a file is passed, import it
    f = if lib.isFunction fn then fn else import fn;

    # find what attrs are shared from expression and package set
    # then override the values by anything passed explicitly through args
    fargs = builtins.intersectAttrs (lib.functionArgs f) autoArgs // args;
  in
    f fargs; # With nix, creation of a derivation is just function application
```

Usage of `callPackage` would look something like this:

```
  # <nixpkgs>/pkgs/top-level/all-packages.nix
  { lib, ... }:

  let
    self = with self; {
      ...

      callPackage = lib.callPackageWith self;

      openssl = callPackage ../libraries/openssl { };
    };
  in self
```

With `callPackage` we only need to explicitly pass an attr set if we need to override the default values that would have been present in the package set.

In nixpkgs, `callPackage` has been extended to include helpful package hints, and thus the complexity has grown, but the underlying intuition has remained the same.

In javascript, `callPackage` would be an example of a curried function, where there's an implicit package set bound to it.

Best Practices

Avoid excessive `with` usage

Although `with` can be useful in small scopes. Doing something such as `with pkgs;` is usually discouraged. This is most dramatic with `pkgs`, in which you will introduce 15,000+ variables into your namespace. Although you may be aware of what is coming from where when you first write the code, this implicit context is much harder to re-create each time the expression is visited in the future. This is compounded with multiple `with` expressions, as later `with`'s will shadow previously defined values.

This is not to say that all usage of `with` is discouraged, it's often encouraged with certain tasks such as defining the `meta` section of a package; as most attributes of a meta section will be pulling from `lib`. So a `meta = with lib; { ... }` can dramatically reduce how many `lib.` need to be explicitly added. Also, it's very common for NixOS modules to use `with lib;` for the whole file as many of the module building blocks are exposed through `lib`.

In general, `with` should be scoped as much as possible:

```
# good
stdenv.mkDerivation {
  ...
  buildInputs = [ openssl ]
    ++ (with xorg; [ libX11 libX11randar xinput ]);
}

# also good, just repetitive
stdenv.mkDerivation {
  ...
  buildInputs = [ openssl xorg.libX11 xorg.libX11randar xorg.xinput ];
}

# discouraged, now all of xorg is exposed everywhere
with xorg;

stdenv.mkDerivation {
  ...
  buildInputs = [ openssl libX11 libX11randar xinput ];
}
```

Common Mistakes

Functions

The space after `:` is required. Without a space, nix will parse the value as an url, and represent it as a string

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

nix-repl> :t x:x
a string
```

Lists

Functions and lists use whitespace to do funcation application, however, list element delimination takes precedence over function application.

For example, if someone were to try and use optional python integration on a package, they may write something like:

```
extraPackages = [
  somePackage.override { withPython = true; }
];
```

In this example, it's an array of two elements, `somePackage.override` is a function, and the other element is an attr set. This is more accurately represented as:

```
extraPackages = [
  (somePackage.override) # type: Attr -> drv
  ({ withPython = true; }) # type: Attr
];
```

The correct usage of this would be:

```
extraPackages = [
  (somePackage.override { withPython = true; }) # type: drv
];
```

Building a Nix Package

Building a package for nix can range from trivial to near impossible. Generally the difference between the two experiences is determined by how many assumptions the build process

makes. Toolchains which have strong integrity guarantees (e.g. lock files) , and allow for offline builds are generally more nix compatible.

Nix is language and toolchain agnostic. Support for many toolchains have been added to nixpkgs, but the nix build environment is very constrained so many `<toolchain>2nix` tools have arisen to try and bridge the gap in expectations.

Simple C program

Many fundamental unix tools are written in C, as it provides many benefits to system programmers. In this section we will cover how to compile and package a simple C application to demonstrate how the nix build process works.

Impure build and install

Given the example C program:

```
$ cat simple.c
#include <stdio.h>

void main() {
  printf("Hello from Nix!");
}
```

The build and installation of which on a traditional FHS system may look like:

```
# build
$ gcc simple.c -o hello_nix
# install
$ sudo cp hello_nix /usr/bin/hello_nix
```

However, let's see how this would be done in nix

Nix build

Implicit to the previous workflow, was the availability of the GNU C Compiler and the usage of the `cp` command. In many package repositories, usage of these tools is near universal; and forms the foundation for how to build most other software.

Although C compilers and GNU's `coreutils` (where `cp` comes from) have their own specific packages in nixpkgs, generally they are aggregated into a pseudo-package called `stdenv` in nixpkgs. The function `stdenv.mkDerivation` provides:

- A nixpkgs-compatible wrapped C compiler (GCC on linux, Clang on MacOS)
- GNU coreutils
- A default "builder" script

`stdenv` will be covered in more detail in the next section.

A nixified version of the build would look like:

```
# simple.nix
let
  pkgs = import <nixpkgs> { };
in
  pkgs.stdenv.mkDerivation {
    name = "hello-nix";

    src = ./.;

    # Use $CC as it allows for stdenv to reference the correct C compiler
    buildPhase = ''
      $CC simple.c -o hello_nix
    '';
  }
```

Nix defaults to a Makefile workflow unless specified otherwise. So stdenv will default to calling `make install` for the `installPhase` which will fail with `No rule to make target 'install'` so we need to also fix how nix will install the package.

```
$ nix-build simple.nix
this derivation will be built:
  /nix/store/dbavzdq1idb0hvwdh7r9gfn2l52kvycf-hello-nix.drv
...
install flags: SHELL=/nix/store/3j918i1nbwhby0y38bn2r438rjhh8f4d-bash-5.1-p16/
bin/bash install
make: *** No rule to make target 'install'.  Stop.
error: builder for '/nix/store/dbavzdq1idb0hvwdh7r9gfn2l52kvycf-hello-nix.drv'
failed with exit code 2;
```

Nix install

The second glaring problem in the old workflow, is that we had a convention as to where to install the executable in /usr/bin/. But installing software in a central location is one the

issues that nix is trying to solve. Instead, nix needs to install files on a per-package basis, thus where we need to install files will change for every package. So how do we know where to install files with nix?

Nix will bind the values defined in the derivation to environment variables inside of the nix build. The default "output" of a package is `out`, which will be bound to the hashed nix store path mentioned in the derivation section.

So an adjusted workflow would be:

```
# build
$ gcc simple.c -o hello_nix
# install
$ mkdir -p $out/bin
$ cp hello_nix $out/bin/hello_nix
```

Extending the example above, the easiest solution would be to write our own `installPhase`. The resulting expression would be:

```
# simple.nix
let
  pkgs = import <nixpkgs> { };
in
  pkgs.stdenv.mkDerivation {
    name = "hello-nix";

    src = ./.;

    buildPhase = ''
      $CC simple.c -o hello_nix
    '';

    installPhase = ''
      mkdir -p $out/bin
      cp hello_nix  $out/bin/hello_nix
    '';
  }
```

Now when we build the package, nix is able to realize it. After which we can use the executable:

```
$ nix-build simple.nix
this derivation will be built:
  /nix/store/9j274i4wckn0ksxpj7asd8vbk67kfz4p-hello-nix.drv
...
/nix/store/giwy9rwzwsdvh86pvdpv37lkwms7xcx9-hello-nix

$ ./result/bin/hello_nix
Hello from Nix!
```

Stdenv

`stdenv` provides a foundation for building C/C++ software with nixpkgs. It includes, but is not limited to containing tools such as: a c compiler and related tools, GNU coreutils, GNU awk, GNU sed, findutils, strip, bash, GNUmake, bzip2, gzip, and many more tools. Stdenv also provides a default "builder.sh" script which will perform the build of a package. The default builder script is comprised of many smaller "phases" which package maintainers can alter slightly as needed. The goal of `stdenv` is to enable most C/C++ + `Makefile` workflows; in theory, if a software package has these installation:

```
./configure # configurePhase, optional
make # buildPhase
make install # installPhase
```

Then the only necessary changes for it to work with `stdenv.mkDerivation` would be the inclusion of `installFlags = [ "PREFIX=$(out)" ];` to communicate where the package should be installed with nix.

Unique qualities of Nixpkgs' Stdenv

Wrapped C Compiler

`stdenv` exposes a wrapped compiler to help communicate nix-specific to the compiler without having to rely on the upstream maintainer to expose such allowances in configuration. For example, let's assume that a package doesn't officially support MacOS, so all testing and building occurs with Linux + GCC. Trying to package this for MacOS might be difficult because the logic may call `gcc` directly, and assume

- Nix differences
    - Wrapped compiler

    - stdenv shell functions

Phases

Build Dependencies

How to add build dependencies for a packages

- Difference between nativeBuildInputs and buildInputs
- Difference between propagated and non-propagated inputs
- Demonstrate some common usage patterns around dependencies

How do packages find dependencies when building?

- They don't, nix attempts to fulfill assumptions made by the toolchain
    - Generally delegated to tooling which specializes in dependency discovery
        - `PKG_CONFIG_PATH` ? for `pkg-config`
        - `CMAKE_MODULE_PATH` ? for `cmake`
        - `PYTHONPATH` for `python` when using `buildPythonPackage`
        - etc.

Explain difference between out, dev, lib, and other outputs

- It's a common use case to reference one or more outputs
- Mention the lib's `getDev` , `getDev` , `getLib` , and `getMan` helpers
- TODO: Link to another section expanding on multi-output derivations

Runtime Dependencies

- Explain how nix finds runtime dependencies (essentially greps for valid /nix/store/... paths)
    - TODO: link to another section on how to reduce closure sizes
    - TODO: link to another section on how to ensure runtime dependencies are correctly picked up
        - E.g. jar files are compressed, but may reference another package which needs to be present on the host
        - Generally this is done by doing `echo ${dependency} > $out/nix-support`
- How to determine a runtime depdency (e.g. `nix-store -q --requisites`)

- How to wrap programs so that certain dependencies are present on PATH or in other ways
- Mention that patching is sometimes required (e.g. python), as there's not always a deterministic way to define how a package will be consumed (e.g. python module importing another)

Building Packages for Other Toolchains

- Introduce the concept of shellhooks
- Provide links to documentation for:
  - Cmake
  - Meson
  - Bazel

Building Nix packages for non-C programming languages

- Almost all support for other programming languages are built upon stdenv.mkDerivation
- Provide links to official documentation on a given programming language

Patching Packages

- Common scenario where a pull request hasn't been merged or a new release hasn't been cut, but a change should be applied to a given package.
- Cover `patches` attr for mkDerivation
- Introduce `fetchpatch` utility
  - Show example
  - Make note that fetchpatch does it's own sanitization, meaning that fetchpatch and `nix-prefetch-url` will generally create different FODs
  - Make note that generally a comment should be added to explain why a patch is being added, and when is it an appropriate time to remove it

Multiple Outputs

- Why multiple outputs
  - Closure size
  - Cross Compilation
- How to Leverage
- Any footguns
  - `placeholder`