Nix - Fundamentals

If you stare into the abyss. . .

Jakub Grzegorz Sokołowski

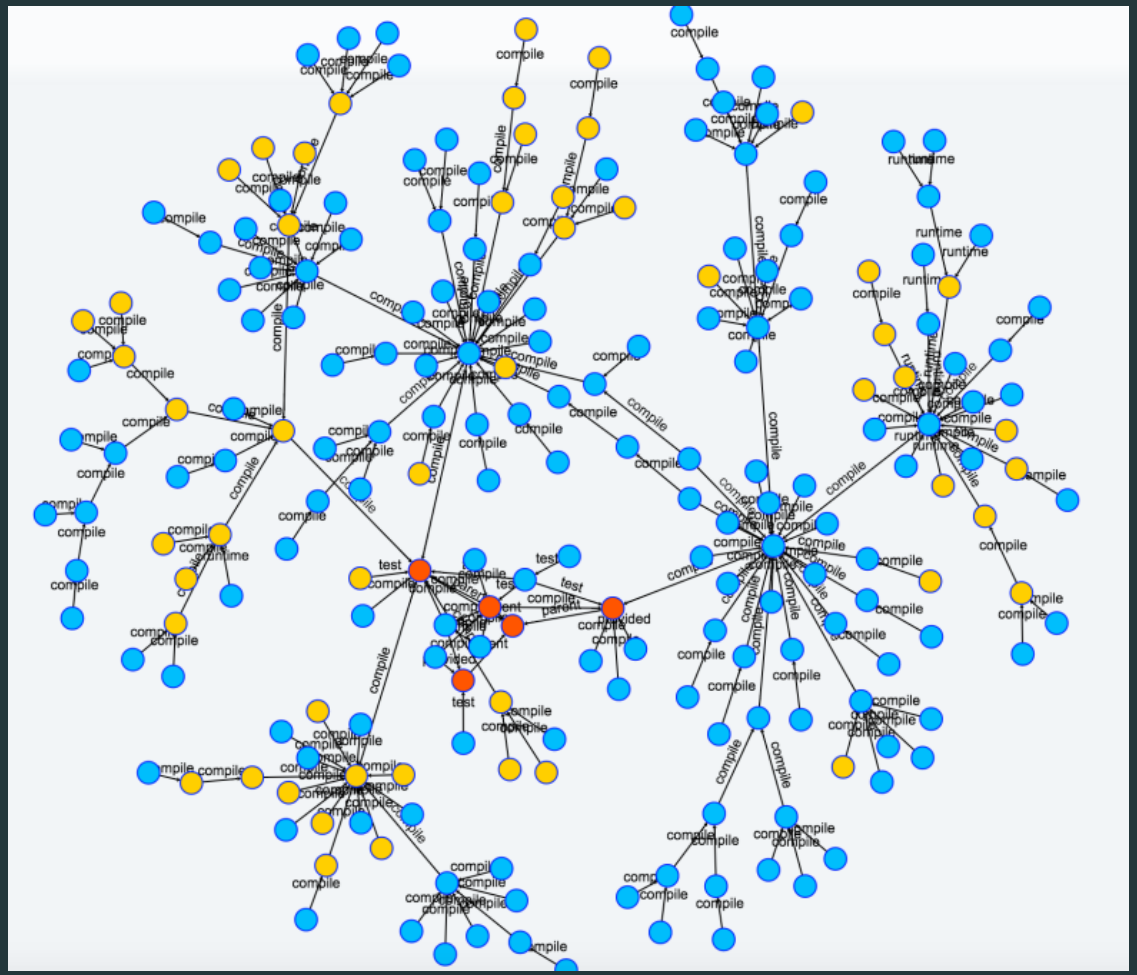
29/05/2020

Contents Overview

What is Nix?

The Purely Functional Package Manager

A way to define software build process and its dependencies in a precise manner.



Syntax uses a subset of Haskell functional programming language.

Basic Syntax - Types

There are eight basic types in Nix:

Name	Example	Description
Null	`null`	It is what it is...
Boolean	`true` and `false`	Yep...
Integer	`123` or `2e12`	Integer ops always return integers
Float	`1.234` or `.27e13`	Float ops always return floats
String	`"some ${type} string"`	Provides interpolation syntax
Lists	`[ 123 ./foo.nix "abc" ]`	Lists can mix types
Sets	`{ a = "Foo"; b = ./Bar; }`	Name/value pairs ending with `;`
Path	`./hello` or `<nixpkgs>`	Square braces resolve from `NIX_PATH`

No dark magic here.

Basic Syntax - Functions

The `:` character is used to indicate function arguments.

```
nix-repl> double = x: x * 2
nix-repl> doube 10
20
```

Here's a simple number adding function:

```
nix-repl> add = x: y: x + y
nix-repl> add 2 2
4
```

Here's the same function taking one **Set** as an argument with defaults:

```
nix-repl> add = { x ? 2, y ? 2 }: x + y
nix-repl> add { x = 1; }
3
```

Passing arguments via a **Set** is the most common way.

Basic Syntax - Expressions

Every Nix file is a Nix expression that can be a function or a value.

But the vast majority of Nix files take the form of a function that accepts an attribute set - essentially an object - as an argument:

```
# Single argument given to the Nix function
# This is an attribute set, it stores keys and values.
{ input ? "default" }:

# Object with three keys: 'aString', 'aList', and 'input'
let
    aSet = {
        # Equivalent to: input = input;
        inherit input;

        # String Interpolation syntax
        aString = "something-${input}";
        aList = ["list" "of" "strings" input];
    };
in
    aSet
```

The `let/in` block allows for defining things you want to use but not return.

Evaluating Functions

The `nix-instantiate` tool can be used to evaluate Nix code:

```
> nix-instantiate --eval -E '1 + 2'
```

```
3
```

You can do the same for our `function.nix` to see its output:

```
> nix-instantiate --eval --strict --arg x 1 function.nix | nixfmt
```

```
{
  input = "default";
  aString = "something-default";
  aList = ["list" "of" "strings" "default"];
}
```

You can also use the `--json` flag to get a more generic output.

Evaluating Functions - Arguments

You can also change the argument value using `--argstr`

```
 > nix-instantiate --eval --strict --argstr input modified function.nix
```

```
{
  input = "modified";
  aString = "something-modified";
  aList = ["list" "of" "strings" "modified"];
}
```

Or `--arg` for more complex arguments like lists of attribute sets.

Importing Files

The way to import another file in Nix is using the `import` keyword:

meta.nix

```
{ pname, version }:

{
    description = "A program that produces a familiar,friendly greeting";
    homepage = "https://www.gnu.org/software/${pname}/manual/";
    changelog = "https://git.gnu.org/${pname}.git/NEWS?h=${version}";
}
```

default.nix

```
let
    pname = "xyz";
    version = "1.2.3";
    src = import ./src.nix { inherit pname version; };
    meta = import ./meta.nix { inherit pname version; };
in stdenv.mkDerivation {
    inherit pname version meta src;
}
```

Importing Files - Implicit Arguments

In order to avoid specifying every single argument we can use `newScope`:

`default.nix`

```
{ pkgs ? import <nixpkgs> { } }:
let
    inherit (pkgs) newScope;

    pname = "xyz";
    version = "1.2.3";
    callPackage = newScope { inherit pname version; }

    # Implicitly passes the pname and version arguments
    src = callPackage ./src.nix { };
    meta = callPackage ./meta.nix { };
in stdenv.mkDerivation {
    inherit pname version meta src;
}
```

The `callPackage` function is heavily used in all of `nixpkgs`, specifically:
https://github.com/NixOS/nixpkgs/blob/master/pkgs/top-level/all-packages.nix

What is a Derivation?

Derivation is a Nix expression describes everything that goes into a package build action:

- Build tools
- Libraries
- Sources
- Build scripts
- Environment variables

Nix tries very hard to ensure that Nix expressions are deterministic: building a Nix expression twice should yield the same result.

Derivations are made with `derivation` primitive or extended `mkDerivation` function.

What is a Derivation? - Example

Here's an example using the GNU Hello package:

```
stdenv.mkDerivation rec {
  pname = "hello";
  version = "2.10";

  src = fetchurl {
    url = "mirror://gnu/hello/${pname}-${version}.tar.gz";
    sha256 = "0ssi1wpaf7plaswqqjwigppsg5fyh99vdlb9kzl7c9lng89ndq1i";
  };

  meta = with stdenv.lib; {
    homepage = "https://www.gnu.org/software/hello/manual/";
    license = licenses.gpl3Plus;
    ...
  };
};
```

How is a Derivation made?

The `mkDerivation` function is an extended version of the built-in `derivation` and it's defined in the `nixpkgs` repository in `pkgs/stdenv/generic/make-derivation.nix`.

It takes arguments like:

- `pname` - Name of the package, user in Nix store path.
- `version` - Self explanatory, also used in the path.
- `src` - Source for the package, usually acquired with `fetchgit` or `fetchurl`.
- `srcs` - Same as `src` except it's a list.
- `buildInputs` - List of packages necessary for build. Tools end up in `PATH`.
- `configureFlags` - Flags for running `./configure` for package.
- `patches` - List of `.patch` files to apply to the package sources.
- `phases` - Way to customize which phases of build run and which don't.
- `meta` - Meta data including maintainers, licenses, and description.
- And many others...

And creates a `.drv` file which contains all the necessary inputs to build the package.

Building a Derivation

The simplest way to build a derivation is to use `nix-build`:

```
 > nix-build --no-out-link --attr pkgs.hello '<nixpkgs>'
/nix/store/fbd9sr5lx1r5r9w3cl4d5p5hgwbhk9jj-hello-2.10
```

You can see it contains the `hello` binary:

```
 > PKG=/nix/store/fbd9sr5lx1r5r9w3cl4d5p5hgwbhk9jj-hello-2.10

 > ls -l ${PKG}/bin
total 41
-r-xr-xr-x 1 root root 37808 Jan  1  1970 hello

 > ${PKG}/bin/hello
Hello, world!

 > ls -l /nix/store/fbd9sr5lx1r5r9w3cl4d5p5hgwbhk9jj-hello-2.10/share
total 10
dr-xr-xr-x  2 root root  3 Jan  1  1970 info
dr-xr-xr-x 44 root root 44 Jan  1  1970 locale
dr-xr-xr-x  3 root root  3 Jan  1  1970 man
```

As well as other things like the manual pages for the package.

Building a Derivation - Inputs

Each derivation has a `*.drv` file which is a specific instance of it's configuration:

```
 > DRV=$(nix-store --query --deriver ${PKG})

 > echo ${DRV}
/nix/store/n2c657xnry51k84zr5gwd7psa9hm92wk-hello-2.10.drv

 > cat ${DRV} | indent | head -n1
Derive ([("out", "/nix/store/fbd9sr5lx1r5r9w3cl4d5p5hgwbhk9jj-hello-2.10"
```

It can be parsed using `nix-store --query`:

```
 > nix-store --query --binding name ${DRV}
hello-2.10

 > nix-store --query --binding stdenv ${DRV}
/nix/store/b4kbdvdyb80r57jhx7665117k75v4jrl-stdenv-linux

 > nix-store --query --binding src ${DRV}
/nix/store/3x7dwzq014bblazs7kq20p9hyzz0qh8g-hello-2.10.tar.gz
```

There's also a new command: `nix show-derivation`

Generic Builder Magic - stdenv

When a derivation is being built an shell environment is created - normally stdenv environment - which provides all the standard build tools like make, patch, or gcc.

Inside of that shell the setup.sh script is sourced which provides a set of Bash functions. Amongst them the most important one is genericBuild.

You can see the stdenv using:

```
 > nix show-derivation nixpkgs.stdenv
{
  "/nix/store/iszb9bc4wbl1xax6a1ppyy73aahdqi3n-stdenv-linux.drv": {
    "outputs": {
      "out": {
        "path":"/nix/store/b4kbdvdyb80r57jhx7665117k75v4jrl-stdenv-linux"
      }
    },
    "inputSrcs": [
      "/nix/store/4ygqr4w06zwcd2kcxa6w3441jijv0pvx-strip.sh",
      "/nix/store/aknix5zw9cj7hd1m3h1d6nnmncl1vkvn-multiple-outputs.sh",
      "/nix/store/c0jwsir691znzb9qrjzpv1jn3w3s4yia-setup.sh",
      "/nix/store/dsyj1sp3h8q2wwi8m6z548rvn3bmm3vc-builder.sh",
      ...
```

https://nixos.org/nixos/nix-pills/fundamentals-of-stdenv.html

Generic Builder Magic - `genericBuilder`

The `genericBuild` function calls **in order** the following phases:

- `unpackPhase` - Unpacks the source provided via `$src` or `$srcs`.
- `patchPhase` - Applies patches from the `$patches` argument. (Optional)
- `configurePhase` - Runs `./configure` with with `$configureFlags`.
- `buildPhase` - Runs `make` with provided `$makeFlags`.
- `checkPhase` - If `$doCheck` is enabled runs `make check`. (Optional)
- `installPhase` - Calls `make install` to put build output into `$out`.
- `fixupPhase` - Stripping binaries, running `patchelf`, fixing paths.
- `installCheckPhase` - Runs `make installcheck` if available. (Optional)
- `distPhase` - Runs `make dist` and copies tarballs. (Optional)

This order chan be changed by setting the `$phases` variable, as well as addition of new phases or removal of existing ones.

Read more about this at: https://nixos.org/nixpkgs/manual/#sec-stdenv-phases

Nix Shell for Building

A way to create the same shell used for building a derivation is using `nix-shell`.

It's crucial to understand that `mkDerivation` arguments are available as env variables.

```
 > nix-shell -A hello '<nixpkgs>'

[nix-shell:~]$ echo $pname $version
hello 2.10

[nix-shell:~]$ echo $src
/nix/store/3x7dwzq014bblazs7kq20p9hyzz0qh8g-hello-2.10.tar.gz

[nix-shell:~]$ echo $stdenv
/nix/store/b4kbdvdyb80r57jhx7665117k75v4jrl-stdenv-linux

[nix-shell:~]$ type genericBuild | head -n2
genericBuild is a function
genericBuild ()

[nix-shell:~]$ grep genericBuild $stdenv/setup
genericBuild() {
```

The `$stdenv` variables contains the `setup.sh` script initialized for this specific build.

Nix Shell for Building - genericBuild

You can use the genericBuild function to run the build for the hello package by hand:

```
[nix-shell:~]$ mkdir /tmp/build
[nix-shell:~]$ cd /tmp/build/
[nix-shell:/tmp/build]$ genericBuild
unpacking sources
unpacking /nix/store/3x7dwzq014bblazs7kq20p9hyzz0qh8g-hello-2.10.tar.gz
source root is hello-2.10
patching sources
configuring
...

[nix-shell:/tmp/build/hello-2.10]$ ls -l hello
-rwxr-xr-x 1 sochan sochan 131032 May 27 20:59 hello

[nix-shell:/tmp/build/hello-2.10]$ ./hello
Hello, world!
```

Making a Custom Nix Shell - Definition

Nix shells are quite powerful, so why not make a Nix shell without a package to build?

That's exactly what `mkShell` is for and it is simply a special type of derivation.

```
# Default value for pkgs by importing default nixpkgs
{ pkgs ? import <nixpkgs> { } }:

pkgs.mkShell {
    # same like any derivation, it needs a name
    name = "strace-shell";
    # if buildInputs have a 'bin' directory they get added to PATH
    buildInputs = with pkgs; [ strace ];
    # gets executed before the shell becomes usable
    shellHook = ''
        export MSG="Hello World!"
        echo $MSG
    '';
}
```

Making a Custom Nix Shell - Usage

By calling `nix-shell` in a directory with the given `shell.nix` we would get:

```
 > nix-shell
Hello World!

[nix-shell:/tmp/x]$ echo $MSG
Hello World!

[nix-shell:/tmp/x]$ which strace
/nix/store/k3srq53i882g96dbbz1k7lg27w1zmljb-strace-5.5/bin/strace
```

And the `mkShell` derivation cannot be built:

```
 > nix-build shell.nix
these derivations will be built:
  /nix/store/yy56hwccgsbglch2x3fkl81lj16nb7rb-my-custom-shell.drv
building '/nix/store/yy56hwccgsbglch2x3fkl81lj16nb7rb-strace-shell.drv'
nobuildPhase

This derivation is not meant to be built, aborting
```

What are Nix Packages

The `<nixpkgs>` special path you see in various places is simply the `nixpkgs` repo: https://github.com/NixOS/nixpkgs

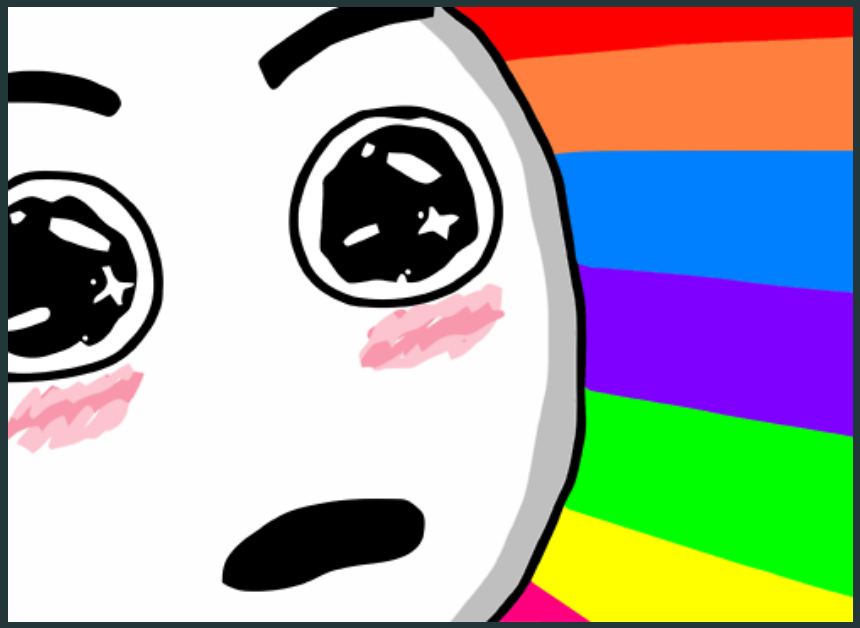
Which is by default found and loaded based on value of `$NIX_PATH` variable.

All of the packages are loaded into one file at: https://github.com/NixOS/nixpkgs/blob/master/pkgs/top-level/all-packages.nix

Which essentially looks like a bunch of `callPackage` calls:

```
fetchzip = callPackage ../build-support/fetchzip { };

fetchCrate = callPackage ../build-support/rust/fetchcrate.nix { };

fetchFromGitHub = callPackage ../build-support/fetchgithub {};

fetchFromBitbucket = callPackage ../build-support/fetchbitbucket {};

fetchFromSavannah = callPackage ../build-support/fetchsavannah {};

fetchFromGitLab = callPackage ../build-support/fetchgitlab {};
```

Final Thoughts



The Good

- Readable definitions of packages
- Hierarchical structure of dependencies
- Easy debugging of package builds
- Ad-hoc customizable shells
- Can be used on any Linux or Darwin
- Operating system built using Nix
  - https://nixos.org/nixos/

The Bad

- Unfamiliar syntax for most
- Massive hard to search docs
- Multitude of similar commands
- Hard to discover libraries

Resources

General

- https://nixos.org/learn.html
- https://nixos.wiki/wiki/Nix_Expression_Language
- https://nixos.org/nixos/nix-pills/index.html
- https://nixos.org/nix/manual/
- https://nixos.org/nixpkgs/manual/
- https://github.com/NixOS/nixpkgs

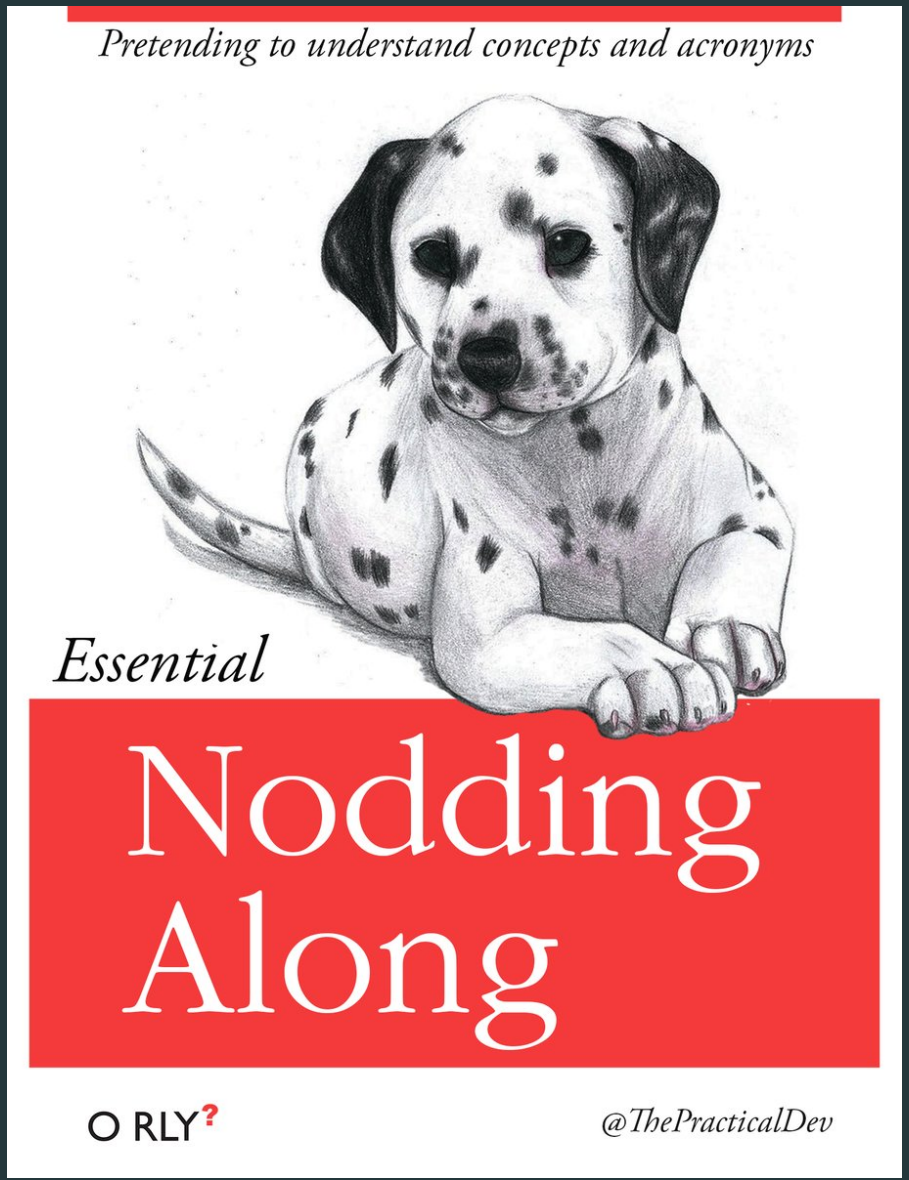
Videos

- Intro to Nix
- Nix: How and Why it Works

Community

- https://discourse.nixos.org/
- IRC - `#nixos` on `freenode.net`

Questions?


Pretending to understand concepts and acronyms
Essential
Nodding Along
O RLY?
@ThePracticalDev