Nixpkgs Manual

Draft (Version 1.0)

Contents

Chapter 1

Introduction

This manual tells you how to write packages for the Nix Packages collection (Nixpkgs). Thus it's for packagers and developers who want to add packages to Nixpkgs. End users are kindly referred to the Nix manual.

This manual does not describe the syntax and semantics of the Nix expression language, which are given in the Nix manual in the chapter on writing Nix expressions. It only describes the facilities provided by Nixpkgs to make writing packages easier, such as the standard build environment (`stdenv`).

Chapter 2

Quick Start to Adding a Package

To add a package to Nixpkgs:

1. Checkout the Nixpkgs source tree:

```
$ svn checkout https://svn.nixos.org/repos/nix/nixpkgs/trunk nixpkgs
$ cd nixpkgs
```

2. Find a good place in the Nixpkgs tree to add the Nix expression for your package. For instance, a library package typically goes into `pkgs/development/libraries/`*`pkgname`*, while a web browser goes into `pkgs/applications/networking/browsers/`*`pkgname`*. See Section 7.3 for some hints on the tree organisation. Create a directory for your package, e.g.

```
$ svn mkdir pkgs/development/libraries/libfoo
```

3. In the package directory, create a Nix expression — a piece of code that describes how to build the package. In this case, it should be a *function* that is called with the package dependencies as arguments, and returns a build of the package in the Nix store. The expression should usually be called `default.nix`.

```
$ emacs pkgs/development/libraries/libfoo/default.nix
$ svn add pkgs/development/libraries/libfoo/default.nix
```

   You can have a look at the existing Nix expressions under `pkgs/` to see how it's done. Here are some good ones:

   - GNU cpio: `pkgs/tools/archivers/cpio/default.nix`. The simplest possible package. The generic builder in `stdenv` does everything for you. It has no dependencies beyond `stdenv`.
   - GNU Hello: `pkgs/applications/misc/hello/ex-2/default.nix`. Also trivial, but it specifies some `meta` attributes which is good practice.
   - GNU Multiple Precision arithmetic library (GMP): `pkgs/development/libraries/gmp/default.nix`. Also done by the generic builder, but has a dependency on `m4`.
   - Pan, a GTK-based newsreader: `pkgs/applications/networking/newsreaders/pan/default.nix`. Has an optional dependency on `gtkspell`, which is only built if `spellCheck` is `true`.
   - Apache HTTPD: `pkgs/servers/http/apache-httpd/default.nix`. A bunch of optional features, variable substitutions in the configure flags, a post-install hook, and miscellaneous hackery.
   - BitTorrent (wxPython-based): `pkgs/tools/networking/p2p/bittorrent/default.nix`. Uses an external build script, which can be useful if you have lots of code that you don't want cluttering up the Nix expression. But external builders are mostly obsolete.
   - Thunderbird: `pkgs/applications/networking/mailreaders/thunderbird/3.x.nix`. Lots of dependencies.
   - JDiskReport, a Java utility: `pkgs/tools/misc/jdiskreport/default.nix` (and the builder). Nixpkgs doesn't have a decent `stdenv` for Java yet so this is pretty ad-hoc.

- XML::Simple, a Perl module: pkgs/top-level/perl-packages.nix (search for the XMLSimple attribute). Most Perl modules are so simple to build that they are defined directly in perl-packages.nix; no need to make a separate file for them.
- Adobe Reader: pkgs/applications/misc/adobe-reader/default.nix. Shows how binary-only packages can be supported. In particular the builder uses **patchelf** to set the RUNPATH and ELF interpreter of the executables so that the right libraries are found at runtime.

Some notes:

- All meta attributes are optional, but it's still a good idea to provide at least the description, homepage and license.
- You can use **nix-prefetch-url** *url* to get the SHA-256 hash of source distributions.
- A list of schemes for mirror:// URLs can be found in pkgs/build-support/fetchurl/mirrors.nix.

The exact syntax and semantics of the Nix expression language, including the built-in function, are described in the Nix manual in the chapter on writing Nix expressions.

4. Add a call to the function defined in the previous step to pkgs/top-level/all-packages.nix with some descriptive name for the variable, e.g. libfoo.

```
$ emacs pkgs/top-level/all-packages.nix
```

The attributes in that file are sorted by category (like "Development / Libraries") that more-or-less correspond to the directory structure of Nixpkgs, and then by attribute name.

5. Test whether the package builds:

```
$ nix-build -A libfoo
```

where libfoo should be the variable name defined in the previous step. You may want to add the flag -K to keep the temporary build directory in case something fails. If the build succeeds, a symlink ./result to the package in the Nix store is created.

6. If you want to install the package into your profile (optional), do

```
$ nix-env -f . -iA libfoo
```

7. Optionally commit the new package (**svn ci**) or send a patch to nix-dev@cs.uu.nl.
8. If you want the TU Delft build farm to build binaries of the package and make them available in the nixpkgs channel, add it to pkgs/top-level/release.nix.

Chapter 3

The Standard Environment

The standard build environment in the Nix Packages collection provides a environment for building Unix packages that does a lot of common build tasks automatically. In fact, for Unix packages that use the standard `./configure; make; make install` build interface, you don't need to write a build script at all; the standard environment does everything automatically. If `stdenv` doesn't do what you need automatically, you can easily customise or override the various build phases.

3.1 Using `stdenv`

To build a package with the standard environment, you use the function `stdenv.mkDerivation`, instead of the primitive built-in function `derivation`, e.g.

```
stdenv.mkDerivation {
  name = "libfoo-1.2.3";
  src = fetchurl {
    url = http://example.org/libfoo-1.2.3.tar.bz2;
    md5 = "e1ec107956b6ddcb0b8b0679367e9ac9";
  };
}
```

(`stdenv` needs to be in scope, so if you write this in a separate Nix expression from `pkgs/all-packages.nix`, you need to pass it as a function argument.) Specifying a `name` and a `src` is the absolute minimum you need to do. Many packages have dependencies that are not provided in the standard environment. It's usually sufficient to specify those dependencies in the `buildInputs` attribute:

```
stdenv.mkDerivation {
  name = "libfoo-1.2.3";
  ...
  buildInputs = [libbar perl ncurses];
}
```

This attribute ensures that the `bin` subdirectories of these packages appear in the `PATH` environment variable during the build, that their `include` subdirectories are searched by the C compiler, and so on. (See Section 3.6 for details.)

Often it is necessary to override or modify some aspect of the build. To make this easier, the standard environment breaks the package build into a number of *phases*, all of which can be overriden or modified individually: unpacking the sources, applying patches, configuring, building, and installing. (There are some others; see Section 3.4.) For instance, a package that doesn't supply a makefile but instead has to be compiled "manually" could be handled like this:

```
stdenv.mkDerivation {
  name = "fnord-4.5";
  ...
  buildPhase = ''
```

```
    gcc foo.c -o foo
  '';
  installPhase = ''
    mkdir -p $out/bin
    cp foo $out/bin
  '';
}
```

(Note the use of `''`-style string literals, which are very convenient for large multi-line script fragments because they don't need escaping of `"` and `\`, and because indentation is intelligently removed.)

There are many other attributes to customise the build. These are listed in Section 3.3.

While the standard environment provides a generic builder, you can still supply your own build script:

```
stdenv.mkDerivation {
  name = "libfoo-1.2.3";
  ...
  builder = ./builder.sh;
}
```

where the builder can do anything it wants, but typically starts with

```
source $stdenv/setup
```

to let `stdenv` set up the environment (e.g., process the `buildInputs`). If you want, you can still use `stdenv`'s generic builder:

```
source $stdenv/setup

buildPhase() {
  echo "... this is my custom build phase ..."
  gcc foo.c -o foo
}

installPhase() {
  mkdir -p $out/bin
  cp foo $out/bin
}

genericBuild
```

3.2 Tools provided by `stdenv`

The standard environment provides the following packages:

- 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** and **bzip2**.
- GNU Make. It has been patched to provide "nested" output that can be fed into the **nix-log2xml** command and **log2html** stylesheet to create a structured, readable output of the build steps performed by Make.
- Bash. This is the shell used for all builders in the Nix Packages collection. Not using **/bin/sh** removes a large source of portability problems.
- The **patch** command.

On Linux, `stdenv` also includes the **patchelf** utility.

3.3 Attributes

VARIABLES AFFECTING `STDENV` INITIALISATION

`NIX_DEBUG` If set, `stdenv` will print some debug information during the build. In particular, the **gcc** and **ld** wrapper scripts will print out the complete command line passed to the wrapped tools.

`buildInputs` A list of dependencies used by `stdenv` to set up the environment for the build. For each dependency *`dir`*, the directory *`dir`*`/bin`, if it exists, is added to the `PATH` environment variable. Other environment variables are also set up via a pluggable mechanism. For instance, if `buildInputs` contains Perl, then the `lib/site_perl` subdirectory of each input is added to the `PERL5LIB` environment variable. See Section 3.6 for details.

`propagatedBuildInputs` Like `buildInputs`, but these dependencies are *propagated*: that is, the dependencies listed here are added to the `buildInputs` of any package that uses *this* package as a dependency. So if package Y has `propagatedBuildInputs = [X]`, and package Z has `buildInputs = [Y]`, then package X will appear in Z's build environment automatically.

3.4 Phases

The generic builder has a number of *phases*. Package builds are split into phases to make it easier to override specific parts of the build (e.g., unpacking the sources or installing the binaries). Furthermore, it allows a nicer presentation of build logs in the Nix build farm.

Each phase can be overriden in its entirety either by setting the environment variable *`name`*`Phase` to a string containing some shell commands to be executed, or by redefining the shell function *`name`*`Phase`. The former is convenient to override a phase from the derivation, while the latter is convenient from a build script.

3.4.1 Controlling phases

There are a number of variables that control what phases are executed and in what order:

VARIABLES AFFECTING PHASE CONTROL

`phases` Specifies the phases. You can change the order in which phases are executed, or add new phases, by setting this variable. If it's not set, the default value is used, which is `$prePhases unpackPhase patchPhase $preConfigurePhases configurePhase $preBuildPhases buildPhase checkPhase $preInstallPhases installPhase fixupPhase $preDistPhases distPhase $postPhases`.

Usually, if you just want to add a few phases, it's more convenient to set one of the variables below (such as `preInstallPhases`), as you then don't specify all the normal phases.

`prePhases` Additional phases executed before any of the default phases.

`preConfigurePhases` Additional phases executed just before the configure phase.

preBuildPhases Additional phases executed just before the build phase.

preInstallPhases Additional phases executed just before the install phase.

preDistPhases Additional phases executed just before the distribution phase.

postPhases Additional phases executed after any of the default phases.

3.4.2 The unpack phase

The unpack phase is responsible for unpacking the source code of the package. The default implementation of `unpackPhase` unpacks the source files listed in the `src` environment variable to the current directory. It supports the following files by default:

Tar files These can optionally be compressed using **gzip** (`.tar.gz`, `.tgz` or `.tar.Z`) or **bzip2** (`.tar.bz2` or `.tbz2`).

Zip files Zip files are unpacked using **unzip**. However, **unzip** is not in the standard environment, so you should add it to `buildInputs` yourself.

Directories in the Nix store These are simply copied to the current directory. The hash part of the file name is stripped, e.g. `/nix/store/1wydxgby13cz...-my-sources` would be copied to `my-sources`.

Additional file types can be supported by setting the `unpackCmd` variable (see below).

VARIABLES CONTROLLING THE UNPACK PHASE

srcs / src The list of source files or directories to be unpacked or copied. One of these must be set.

sourceRoot After running `unpackPhase`, the generic builder changes the current directory to the directory created by unpacking the sources. If there are multiple source directories, you should set `sourceRoot` to the name of the intended directory.

setSourceRoot Alternatively to setting `sourceRoot`, you can set `setSourceRoot` to a shell command to be evaluated by the unpack phase after the sources have been unpacked. This command must set `sourceRoot`.

preUnpack Hook executed at the start of the unpack phase.

postUnpack Hook executed at the end of the unpack phase.

dontMakeSourcesWritable If set to `1`, the unpacked sources are *not* made writable. By default, they are made writable to prevent problems with read-only sources. For example, copied store directories would be read-only without this.

unpackCmd The unpack phase evaluates the string `$unpackCmd` for any unrecognised file. The path to the current source file is contained in the `curSrc` variable.

3.4.3 The patch phase

The patch phase applies the list of patches defined in the `patches` variable.

VARIABLES CONTROLLING THE PATCH PHASE

patches The list of patches. They must be in the format accepted by the **patch** command, and may optionally be compressed using **gzip** (`.gz`) or **bzip2** (`.bz2`).

patchFlags Flags to be passed to **patch**. If not set, the argument `-p1` is used, which causes the leading directory component to be stripped from the file names in each patch.

prePatch Hook executed at the start of the patch phase.

postPatch Hook executed at the end of the patch phase.

3.4.4 The configure phase

The configure phase prepares the source tree for building. The default `configurePhase` runs `./configure` (typically an Autoconf-generated script) if it exists.

VARIABLES CONTROLLING THE CONFIGURE PHASE

`configureScript` The name of the configure script. It defaults to `./configure` if it exists; otherwise, the configure phase is skipped. This can actually be a command (like `perl ./Configure.pl`).

`configureFlags` Additional arguments passed to the configure script.

`configureFlagsArray` A shell array containing additional arguments passed to the configure script. You must use this instead of `configureFlags` if the arguments contain spaces.

`dontAddPrefix` By default, the flag `--prefix=$prefix` is added to the configure flags. If this is undesirable, set this variable to a non-empty value.

`prefix` The prefix under which the package must be installed, passed via the `--prefix` option to the configure script. It defaults to `$out`.

`dontAddDisableDepTrack` By default, the flag `--disable-dependency-tracking` is added to the configure flags to speed up Automake-based builds. If this is undesirable, set this variable to a non-empty value.

`dontFixLibtool` By default, the configure phase applies some special hackery to all files called `ltmain.sh` before running the configure script in order to improve the purity of Libtool-based packages[1]. If this is undesirable, set this variable to a non-empty value.

`preConfigure` Hook executed at the start of the configure phase.

`postConfigure` Hook executed at the end of the configure phase.

3.4.5 The build phase

The build phase is responsible for actually building the package (e.g. compiling it). The default `buildPhase` simply calls **make** if a file named `Makefile`, `makefile` or `GNUmakefile` exists in the current directory (or the `makefile` is explicitly set); otherwise it does nothing.

VARIABLES CONTROLLING THE BUILD PHASE

`makefile` The file name of the Makefile.

`makeFlags` Additional flags passed to **make**. These flags are also used by the default install and check phase. For setting make flags specific to the build phase, use `buildFlags` (see below).

`makeFlagsArray` A shell array containing additional arguments passed to **make**. You must use this instead of `makeFlags` if the arguments contain spaces, e.g.

```
makeFlagsArray=(CFLAGS="-O0 -g" LDFLAGS="-lfoo -lbar")
```

Note that shell arrays cannot be passed through environment variables, so you cannot set `makeFlagsArray` in a derivation attribute (because those are passed through environment variables): you have to define them in shell code.

`buildFlags / buildFlagsArray` Additional flags passed to **make**. Like `makeFlags` and `makeFlagsArray`, but only used by the build phase.

`preBuild` Hook executed at the start of the build phase.

`postBuild` Hook executed at the end of the build phase.

You can set flags for **make** through the `makeFlags` variable.

Before and after running **make**, the hooks `preBuild` and `postBuild` are called, respectively.

[1] It clears the `sys_lib_*search_path` variables in the Libtool script to prevent Libtool from using libraries in `/usr/lib` and such.

3.4.6 The check phase

The check phase checks whether the package was built correctly by running its test suite. The default `checkPhase` calls **make check**, but only if the `doCheck` variable is enabled.

VARIABLES CONTROLLING THE CHECK PHASE

`doCheck` If set to a non-empty string, the check phase is executed, otherwise it is skipped (default). Thus you should set

```
doCheck = true;
```

in the derivation to enable checks.

`makeFlags` / `makeFlagsArray` / `makefile` See the build phase for details.

`checkTarget` The make target that runs the tests. Defaults to `check`.

`checkFlags` / `checkFlagsArray` Additional flags passed to **make**. Like `makeFlags` and `makeFlagsArray`, but only used by the check phase.

`preCheck` Hook executed at the start of the check phase.

`postCheck` Hook executed at the end of the check phase.

3.4.7 The install phase

The install phase is responsible for installing the package in the Nix store under `out`. The default `installPhase` creates the directory `$out` and calls **make install**.

VARIABLES CONTROLLING THE CHECK PHASE

`makeFlags` / `makeFlagsArray` / `makefile` See the build phase for details.

`installTargets` The make targets that perform the installation. Defaults to `install`. Example:

```
installTargets = "install-bin install-doc";
```

`installFlags` / `installFlagsArray` Additional flags passed to **make**. Like `makeFlags` and `makeFlagsArray`, but only used by the install phase.

`preInstall` Hook executed at the start of the install phase.

`postInstall` Hook executed at the end of the install phase.

3.4.8 The fixup phase

The fixup phase performs some (Nix-specific) post-processing actions on the files installed under `$out` by the install phase. The default `fixupPhase` does the following:

- It moves the `man/`, `doc/` and `info/` subdirectories of `$out` to `share/`.
- It strips libraries and executables of debug information.
- On Linux, it applies the **patchelf** command to ELF executables and libraries to remove unused directories from the `RPATH` in order to prevent unnecessary runtime dependencies.
- It rewrites the interpreter paths of shell scripts to paths found in `PATH`. E.g., `/usr/bin/perl` will be rewritten to `/nix/store/`*`some-perl`*`/bin/perl` found in `PATH`.

VARIABLES CONTROLLING THE CHECK PHASE

dontStrip If set, libraries and executables are not stripped. By default, they are.

stripAllList List of directories to search for libraries and executables from which *all* symbols should be stripped. By default, it's empty. Stripping all symbols is risky, since it may remove not just debug symbols but also ELF information necessary for normal execution.

stripAllFlags Flags passed to the **strip** command applied to the files in the directories listed in `stripAllList`. Defaults to `-s` (i.e. `--strip-all`).

stripDebugList List of directories to search for libraries and executables from which only debugging-related symbols should be stripped. It defaults to `lib bin sbin`.

stripDebugFlags Flags passed to the **strip** command applied to the files in the directories listed in `stripDebugList`. Defaults to `-S` (i.e. `--strip-debug`).

dontPatchELF If set, the **patchelf** command is not used to remove unnecessary `RPATH` entries. Only applies to Linux.

dontPatchShebangs If set, scripts starting with `#!` do not have their interpreter paths rewritten to paths in the Nix store.

forceShare The list of directories that must be moved from `$out` to `$out/share`. Defaults to `man doc info`.

setupHook A package can export a setup hook by setting this variable. The setup hook, if defined, is copied to `$out/nix-support/setup-hook`. Environment variables are then substituted in it using `substituteAll`.

preFixup Hook executed at the start of the fixup phase.

postFixup Hook executed at the end of the fixup phase.

3.4.9 The distribution phase

The distribution phase is intended to produce a source distribution of the package. The default `distPhase` first calls **make dist**, then it copies the resulting source tarballs to `$out/tarballs/`. This phase is only executed if the attribute `doDist` is set.

VARIABLES CONTROLLING THE DISTRIBUTION PHASE

distTarget The make target that produces the distribution. Defaults to `dist`.

distFlags / distFlagsArray Additional flags passed to **make**.

tarballs The names of the source distribution files to be copied to `$out/tarballs/`. It can contain shell wildcards. The default is `*.tar.gz`.

dontCopyDist If set, no files are copied to `$out/tarballs/`.

preDist Hook executed at the start of the distribution phase.

postDist Hook executed at the end of the distribution phase.

3.5 Shell functions

The standard environment provides a number of useful functions.

substitute *infile outfile subs* Performs string substitution on the contents of *infile*, writing the result to *outfile*. The substitutions in *subs* are of the following form:

- **--replace *s1 s2*** Replace every occurence of the string *s1* by *s2*.
- **--subst-var *varName*** Replace every occurence of `@`*varName*`@` by the contents of the environment variable *varName*. This is useful for generating files from templates, using `@...@` in the template as placeholders.
- **--subst-var-by *varName s*** Replace every occurence of `@`*varName*`@` by the string *s*.

Example:

```
substitute ./foo.in ./foo.out \
    --replace /usr/bin/bar $bar/bin/bar \
    --replace "a string containing spaces" "some other text" \
    --subst-var someVar
```

`substitute` is implemented using the **replace** command. Unlike with the **sed** command, you don't have to worry about escaping special characters. It supports performing substitutions on binary files (such as executables), though there you'll probably want to make sure that the replacement string is as long as the replaced string.

substituteInPlace ***file subs*** Like `substitute`, but performs the substitutions in place on the file *file*.

substituteAll ***infile outfile*** Replaces every occurence of `@`*varName*`@`, where *varName* is any environment variable, in *infile*, writing the result to *outfile*. For instance, if *infile* has the contents

```
#! @bash@/bin/sh
PATH=@coreutils@/bin
echo @foo@
```

and the environment contains `bash=/nix/store/bmwp0q28cf21...-bash-3.2-p39` and `coreutils=/nix/store/68afga4khv0w...-coreutils-6.12`, but does not contain the variable `foo`, then the output will be

```
#! /nix/store/bmwp0q28cf21...-bash-3.2-p39/bin/sh
PATH=/nix/store/68afga4khv0w...-coreutils-6.12/bin
echo @foo@
```

That is, no substitution is performed for undefined variables.

substituteAllInPlace ***file*** Like `substituteAll`, but performs the substitutions in place on the file *file*.

stripHash ***path*** Strips the directory and hash part of a store path, and prints (on standard output) only the name part. For instance, `stripHash /nix/store/68afga4khv0w...-coreutils-6.12` print `coreutils-6.12`.

3.6 Package setup hooks

The following packages provide a setup hook:

GCC wrapper Adds the `include` subdirectory of each build input to the `NIX_CFLAGS_COMPILE` environment variable, and the `lib` and `lib64` subdirectories to `NIX_LDFLAGS`.

Perl Adds the `lib/site_perl` subdirectory of each build input to the `PERL5LIB` environment variable.

Python Adds the `lib/python2.5/site-packages` subdirectory of each build input to the `PYTHONPATH` environment variable.

> **Note**
> This should be generalised: the Python version shouldn't be hard-coded.

pkg-config Adds the `lib/pkgconfig` and `share/pkgconfig` subdirectories of each build input to the `PKG_CONFIG_PATH` environment variable.

Automake Adds the `share/aclocal` subdirectory of each build input to the `ACLOCAL_PATH` environment variable.

libxml2 Adds every file named `catalog.xml` found under the `xml/dtd` and `xml/xsl` subdirectories of each build input to the `XML_CATALOG_FILES` environment variable.

teTeX / TeX Live Adds the `share/texmf-nix` subdirectory of each build input to the `TEXINPUTS` environment variable.

Qt Sets the `QTDIR` environment variable to Qt's path.

GHC Creates a temporary package database and registers every Haskell build input in it (TODO: how?).

3.7 Purity in Nixpkgs

[measures taken to prevent dependencies on packages outside the store, and what you can do to prevent them]

GCC doesn't search in locations such as `/usr/include`. In fact, attempts to add such directories through the `-I` flag are filtered out. Likewise, the linker (from GNU binutils) doesn't search in standard locations such as `/usr/lib`. Programs built on Linux are linked against a GNU C Library that likewise doesn't search in the default system locations.

Chapter 4

Meta-attributes

Nix packages can declare *meta-attributes* that contain information about a package such as a description, its homepage, its license, and so on. For instance, the GNU Hello package has a `meta` declaration like this:

```
meta = {
  description = "A program that produces a familiar, friendly greeting";
  longDescription = ''
    GNU Hello is a program that prints "Hello, world!" when you run it.
    It is fully customizable.
  '';
  homepage = http://www.gnu.org/software/hello/manual/;
  license = "GPLv3+";
};
```

Meta-attributes are not passed to the builder of the package. Thus, a change to a meta-attribute doesn't trigger a recompilation of the package. The value of a meta-attribute must a string.

The meta-attributes of a package can be queried from the command-line using **nix-env**:

```
$ nix-env -qa hello --meta --xml
<?xml version='1.0' encoding='utf-8'?>
<items>
  <item attrPath="hello" name="hello-2.3" system="i686-linux">
    <meta name="description" value="A program that produces a familiar, friendly greeting"  ↩
        />
    <meta name="homepage" value="http://www.gnu.org/software/hello/manual/" />
    <meta name="license" value="GPLv3+" />
    <meta name="longDescription" value="GNU Hello is a program that prints &amp;quot;Hello,  ↩
        world!&amp;quot; when you run it.&amp;#xA;It is fully customizable.&amp;#xA;" />
  </item>
</items>
```

nix-env knows about the `description` field specifically:

```
$ nix-env -qa hello --description
hello-2.3  A program that produces a familiar, friendly greeting
```

4.1 Standard meta-attributes

The following meta-attributes have a standard interpretation:

description A short (one-line) description of the package. This is shown by **nix-env -q --description** and also on the Nixpkgs release pages.

Don't include a period at the end. Don't include newline characters. Capitalise the first character. For brevity, don't repeat the name of package — just describe what it does.

Wrong: `"libpng is a library that allows you to decode PNG images."`

Right: `"A library for decoding PNG images"`

longDescription An arbitrarily long description of the package.

homepage The package's homepage. Example: `http://www.gnu.org/software/hello/manual/`

license The license for the package. See below for the allowed values.

maintainers A list of names and e-mail addresses of the maintainers of this Nix expression, e.g. `["Alice <alice@example.org>" "Bob <bob@example.com>"]`. If you are the maintainer of multiple packages, you may want to add yourself to `pkgs/lib/maintainers.nix` and write something like `[stdenv.lib.maintainers.alice stdenv.lib.maintainers.bob]`.

priority The *priority* of the package, used by **nix-env** to resolve file name conflicts between packages. See the Nix manual page for **nix-env** for details. Example: `"10"` (a low-priority package).

4.2 Licenses

Note
This is just a first attempt at standardising the license attribute.

The `meta.license` attribute must be one of the following:

GPL GNU General Public License; version not specified.

GPLv2 GNU General Public License, version 2.

GPLv2+ GNU General Public License, version 2 or higher.

GPLv3 GNU General Public License, version 3.

GPLv3+ GNU General Public License, version 3 or higher.

bsd Catch-all for licenses that are essentially similar to the original BSD license with the advertising clause removed, i.e. permissive non-copyleft free software licenses. This includes the X11 ("MIT") License.

free Catch-all for free software licenses not listed above.

free-copyleft Catch-all for free, copyleft software licenses not listed above.

free-non-copyleft Catch-all for free, non-copyleft software licenses not listed above.

unfree-redistributable Unfree package that can be redistributed in binary form. That is, it's legal to redistribute the *output* of the derivation. This means that the package can be included in the Nixpkgs channel.

Sometimes proprietary software can only be redistributed unmodified. Make sure the builder doesn't actually modify the original binaries; otherwise we're breaking the license. For instance, the NVIDIA X11 drivers can be redistributed unmodified, but our builder applies **patchelf** to make them work. Thus, its license is `unfree` and it cannot be included in the Nixpkgs channel.

unfree Unfree package that cannot be redistributed. You can build it yourself, but you cannot redistribute the output of the derivation. Thus it cannot be included in the Nixpkgs channel.

unfree-redistributable-firmware This package supplies unfree, redistributable firmware. This is a separate value from `unfree-redistributable` because not everybody cares whether firmware is free.

Chapter 5

Support for specific programming languages

The standard build environment makes it easy to build typical Autotools-based packages with very little code. Any other kind of package can be accomodated by overriding the appropriate phases of `stdenv`. However, there are specialised functions in Nixpkgs to easily build packages for other programming languages, such as Perl or Haskell. These are described in this chapter.

5.1 Perl

Nixpkgs provides a function `buildPerlPackage`, a generic package builder function for any Perl package that has a standard `Makefile.PL`. It's implemented in `pkgs/development/perl-modules/generic`.

Perl packages from CPAN are defined in `pkgs/perl-packages.nix`, rather than `pkgs/all-packages.nix`. Most Perl packages are so straight-forward to build that they are defined here directly, rather than having a separate function for each package called from `perl-packages.nix`. However, more complicated packages should be put in a separate file, typically in `pkgs/development/perl-modules`. Here is an example of the former:

```
ClassC3 = buildPerlPackage rec {
  name = "Class-C3-0.21";
  src = fetchurl {
    url = "mirror://cpan/authors/id/F/FL/FLORA/${name}.tar.gz";
    sha256 = "1bl8z095y4js66pwxnm7s853pi9czala4sqc743fdlnk27kq94gz";
  };
};
```

Note the use of `mirror://cpan/`, and the `${name}` in the URL definition to ensure that the name attribute is consistent with the source that we're actually downloading. Perl packages are made available in `all-packages.nix` through the variable `perlPackages`. For instance, if you have a package that needs `ClassC3`, you would typically write

```
foo = import ../path/to/foo.nix {
  inherit stdenv fetchurl ...;
  inherit (perlPackages) ClassC3;
};
```

in `all-packages.nix`. You can test building a Perl package as follows:

```
$ nix-build -A perlPackages.ClassC3
```

`buildPerlPackage` adds `perl-` to the start of the name attribute, so the package above is actually called `perl-Class-C3-0.21`. So to install it, you can say:

```
$ nix-env -i perl-Class-C3
```

(Of course you can also install using the attribute name: `nix-env -i -A perlPackages.ClassC3`.)

So what does `buildPerlPackage` do? It does the following:

1. In the configure phase, it calls `perl Makefile.PL` to generate a Makefile. You can set the variable `makeMakerFlags` to pass flags to `Makefile.PL`
2. It adds the contents of the `PERL5LIB` environment variable to `#! .../bin/perl` line of Perl scripts as `-Idir` flags. This ensures that a script can find its dependencies.
3. In the fixup phase, it writes the propagated build inputs (`propagatedBuildInputs`) to the file `$out/nix-support/propagated-user-env-packages`. **nix-env** recursively installs all packages listed in this file when you install a package that has it. This ensures that a Perl package can find its dependencies.

`buildPerlPackage` is built on top of `stdenv`, so everything can be customised in the usual way. For instance, the `BerkeleyDB` module has a `preConfigure` hook to generate a configuration file used by `Makefile.PL`:

```
{buildPerlPackage, fetchurl, db4}:

buildPerlPackage rec {
  name = "BerkeleyDB-0.36";

  src = fetchurl {
    url = "mirror://cpan/authors/id/P/PM/PMQS/${name}.tar.gz";
    sha256 = "07xf50riarb60l1h6m2dqmql8q5dij619712fsgw7ach04d8g3z1";
  };

  preConfigure = ''
    echo "LIB = ${db4}/lib" > config.in
    echo "INCLUDE = ${db4}/include" >> config.in
  '';
}
```

Dependencies on other Perl packages can be specified in the `buildInputs` and `propagatedBuildInputs` attributes. If something is exclusively a build-time dependency, use `buildInputs`; if it's (also) a runtime dependency, use `propagatedBuildInputs`. For instance, this builds a Perl module that has runtime dependencies on a bunch of other modules:

```
ClassC3Componentised = buildPerlPackage rec {
  name = "Class-C3-Componentised-1.0004";
  src = fetchurl {
    url = "mirror://cpan/authors/id/A/AS/ASH/${name}.tar.gz";
    sha256 = "0xql73jkcdbq4q9m0b0rnca6nrlvf5hyzy8is0crdk65bynvs8q1";
  };
  propagatedBuildInputs = [
    ClassC3 ClassInspector TestException MROCompat
  ];
};
```

5.2 Python

Python packages that use `setuptools`, which many Python packages do nowadays, can be built very simply using the `buildPythonPackage` function. This function is implemented in `pkgs/development/python-modules/generic/default.nix` and works similarly to `buildPerlPackage`. (See Section 5.1 for details.)

Python packages that use `buildPythonPackage` are defined in `pkgs/top-level/python-packages.nix`. Most of them are simple. For example:

```
twisted = buildPythonPackage {
  name = "twisted-8.1.0";

  src = fetchurl {
    url = http://tmrc.mit.edu/mirror/twisted/Twisted/8.1/Twisted-8.1.0.tar.bz2;
    sha256 = "0q25zbr4xzknaghha72mq57kh53qw1bf8csgp63pm9sfi72qhirl";
```

```
  };

  propagatedBuildInputs = [ pkgs.ZopeInterface ];

  meta = {
    homepage = http://twistedmatrix.com/;
    description = "Twisted, an event-driven networking engine written in Python";
    license = "MIT";
  };
};
```

Chapter 6

Package Notes

This chapter contains information about how to use and maintain the Nix expressions for a number of specific packages, such as the Linux kernel or X.org.

6.1 Linux kernel

The Nix expressions to build the Linux kernel are in `pkgs/os-specific/linux/kernel`.

The function that builds the kernel has an argument `kernelPatches` which should be a list of `{name, patch, extraConfig}` attribute sets, where `name` is the name of the patch (which is included in the kernel's `meta.description` attribute), `patch` is the patch itself (possibly compressed), and `extraConfig` (optional) is a string specifying extra options to be concatenated to the kernel configuration file (`.config`).

The kernel derivation exports an attribute `features` specifying whether optional functionality is or isn't enabled. This is used in NixOS to implement kernel-specific behaviour. For instance, if the kernel has the `iwlwifi` feature (i.e. has built-in support for Intel wireless chipsets), then NixOS doesn't have to build the external `iwlwifi` package:

```
modulesTree = [kernel]
  ++ pkgs.lib.optional (!kernel.features ? iwlwifi) kernelPackages.iwlwifi
  ++ ...;
```

How to add a new (major) version of the Linux kernel to Nixpkgs:

1. Copy (**svn cp**) the old Nix expression (e.g. `linux-2.6.21.nix`) to the new one (e.g. `linux-2.6.22.nix`) and update it.
2. Add the new kernel to `all-packages.nix` (e.g., create an attribute `kernel_2_6_22`).
3. Now we're going to update the kernel configuration. First unpack the kernel. Then for each supported platform (`i686`, `x86_64`, `uml`) do the following:
   (a) Make an **svn copy** from the old config (e.g. `config-2.6.21-i686-smp`) to the new one (e.g. `config-2.6.22-i686-smp`).
   (b) Copy the config file for this platform (e.g. `config-2.6.22-i686-smp`) to `.config` in the kernel source tree.
   (c) Run `make oldconfig ARCH=`*`{i386,x86_64,um}`* and answer all questions. (For the uml configuration, also add `SHELL=bash`.) Make sure to keep the configuration consistent between platforms (i.e. don't enable some feature on `i686` and disable it on `x86_64`).
   (d) If needed you can also run `make menuconfig`:

   ```
   $ nix-env -i ncurses
   $ export NIX_CFLAGS_LINK=-lncurses
   $ make menuconfig ARCH=arch
   ```

(e) Make sure that `CONFIG_FB_TILEBLITTING` is *not set* (otherwise **fbsplash** won't work). This option has a tendency to be enabled as a side-effect of other options. If it is, investigate why (there's probably another option that forces it to be on) and fix it.

(f) Copy `.config` over the new config file (e.g. `config-2.6.22-i686-smp`).

4. Test building the kernel: `nix-build -A kernel_2_6_22`. If it compiles, ship it! For extra credit, try booting NixOS with it.

5. It may be that the new kernel requires updating the external kernel modules and kernel-dependent packages listed in the `kernelPackagesFor` function in `all-packages.nix` (such as the NVIDIA drivers, AUFS, splashutils, etc.). If the updated packages aren't backwards compatible with older kernels, you need to keep the older versions and use some conditionals. For example, new kernels require splashutils 1.5 while old kernel require 1.3, so `kernelPackagesFor` says:

```
splashutils =
  if kernel.features ? fbSplash then splashutils_13 else
  if kernel.features ? fbConDecor then splashutils_15 else
  null;

splashutils_13 = ...;
splashutils_15 = ...;
```

6.2 X.org

The Nix expressions for the X.org packages reside in `pkgs/servers/x11/xorg/default.nix`. This file is automatically generated from lists of tarballs in an X.org release. As such it should not be modified directly; rather, you should modify the lists, the generator script or the file `pkgs/servers/x11/xorg/overrides.nix`, in which you can override or add to the derivations produced by the generator.

The generator is invoked as follows:

```
$ cd pkgs/servers/x11/xorg
$ cat tarballs-7.5.list extra.list old.list \
  | perl ./generate-expr-from-tarballs.pl
```

For each of the tarballs in the `.list` files, the script downloads it, unpacks it, and searches its `configure.ac` and `*.pc.in` files for dependencies. This information is used to generate `default.nix`. The generator caches downloaded tarballs between runs. Pay close attention to the `NOT FOUND:` *`name`* messages at the end of the run, since they may indicate missing dependencies. (Some might be optional dependencies, however.)

A file like `tarballs-7.5.list` contains all tarballs in a X.org release. It can be generated like this:

```
$ export i="mirror://xorg/X11R7.4/src/everything/"
$ cat $(PRINT_PATH=1 nix-prefetch-url $i | tail -n 1) \
  | perl -e 'while (<>) { if (/(href|HREF)="([^"]*.bz2)"/) { print "$ENV{'i'}$2\n"; }; }' \
  | sort > tarballs-7.4.list
```

`extra.list` contains libraries that aren't part of X.org proper, but are closely related to it, such as `libxcb`. `old.list` contains some packages that were removed from X.org, but are still needed by some people or by other packages (such as `imake`).

If the expression for a package requires derivation attributes that the generator cannot figure out automatically (say, `patches` or a `postInstall` hook), you should modify `pkgs/servers/x11/xorg/overrides.nix`.

Chapter 7

Coding conventions

7.1 Syntax

- Use 2 spaces of indentation per indentation level in Nix expressions, 4 spaces in shell scripts.
- Do not use tab characters, i.e. configure your editor to use soft tabs. For instance, use `(setq-default indent-tabs-mode nil)` in Emacs. Everybody has different tab settings so it's asking for trouble.
- Use `lowerCamelCase` for variable names, not `UpperCamelCase`. TODO: naming of attributes in `all-packages.nix`?
- Function calls with attribute set arguments are written as

```
foo {
  arg = ...;
}
```

  not

```
foo
{
  arg = ...;
}
```

  Also fine is

```
foo { arg = ...; }
```

  if it's a short call.

- In attribute sets or lists that span multiple lines, the attribute names or list elements should be aligned:

```
# A long list.
list =
  [ elem1
    elem2
    elem3
  ];

# A long attribute set.
attrs =
  { attr1 = short_expr;
    attr2 =
      if true then big_expr else big_expr;
  };
```

```
# Alternatively:
attrs = {
  attr1 = short_expr;
  attr2 =
    if true then big_expr else big_expr;
};
```

- Short lists or attribute sets can be written on one line:

```
# A short list.
list = [ elem1 elem2 elem3 ];

# A short set.
attrs = { x = 1280; y = 1024; };
```

- Breaking in the middle of a function argument can give hard-to-read code, like

```
someFunction { x = 1280;
  y = 1024; } otherArg
  yetAnotherArg
```

  (especially if the argument is very large, spanning multiple lines).

  Better:

```
someFunction
  { x = 1280; y = 1024; }
  otherArg
  yetAnotherArg
```

  or

```
let res = { x = 1280; y = 1024; };
in someFunction res otherArg yetAnotherArg
```

- The bodies of functions, asserts, and withs are not indented to prevent a lot of superfluous indentation levels, i.e.

```
{ arg1, arg2 }:
assert system == "i686-linux";
stdenv.mkDerivation { ...
```

  not

```
{ arg1, arg2 }:
  assert system == "i686-linux";
    stdenv.mkDerivation { ...
```

- Function formal arguments are written as:

```
{ arg1, arg2, arg3 }:
```

  but if they don't fit on one line they're written as:

```
{ arg1, arg2, arg3
, arg4, ...
, # Some comment...
  argN
}:
```

- Functions should list their expected arguments as precisely as possible. That is, write

```
{ stdenv, fetchurl, perl }: ...
```

instead of

```
args: with args; ...
```

or

```
{ stdenv, fetchurl, perl, ... }: ...
```

For functions that are truly generic in the number of arguments (such as wrappers around `mkDerivation`) that have some required arguments, you should write them using an `@`-pattern:

```
{ stdenv, doCoverageAnalysis ? false, ... } @ args:

stdenv.mkDerivation (args // {
  ... if doCoverageAnalysis then "bla" else "" ...
})
```

instead of

```
args:

args.stdenv.mkDerivation (args // {
  ... if args ? doCoverageAnalysis && args.doCoverageAnalysis then "bla" else "" ...
})
```

7.2 Package naming

In Nixpkgs, there are generally three different names associated with a package:

- The `name` attribute of the derivation (excluding the version part). This is what most users see, in particular when using **nix-env**.
- The variable name used for the instantiated package in `all-packages.nix`, and when passing it as a dependency to other functions. This is what Nix expression authors see. It can also be used when installing using **nix-env -iA**.
- The filename for (the directory containing) the Nix expression.

Most of the time, these are the same. For instance, the package `e2fsprogs` has a `name` attribute "`e2fsprogs-`*`version`*", is bound to the variable name `e2fsprogs` in `all-packages.nix`, and the Nix expression is in `pkgs/os-specific/linux/e2fsprogs/default.nix`. However, identifiers in the Nix language don't allow certain characters (e.g. dashes), so sometimes a different variable name should be used. For instance, the `module-init-tools` package is bound to the `module_init_tools` variable in `all-packages.nix`.

There are a few naming guidelines:

- Generally, try to stick to the upstream package name.
- Don't use uppercase letters in the `name` attribute — e.g., "`mplayer-1.0rc2`" instead of "`MPlayer-1.0rc2`".
- The version part of the `name` attribute *must* start with a digit (following a dash) — e.g., "`hello-0.3-pre-r3910`" instead of "`hello-svn-r3910`", as the latter would be seen as a package named `hello-svn` by **nix-env**.
- Dashes in the package name should be changed to underscores in variable names, rather than to camel case — e.g., `module_init_tools` instead of `moduleInitTools`.
- If there are multiple versions of a package, this should be reflected in the variable names in `all-packages.nix`, e.g. `hello_0_3` and `hello_0_4`. If there is an obvious "default" version, make an attribute like `hello = hello_0_4;`.

7.3 File naming and organisation

Names of files and directories should be in lowercase, with dashes between words — not in camel case. For instance, it should be `all-packages.nix`, not `allPackages.nix` or `AllPackages.nix`.

7.3.1 Hierachy

Each package should be stored in its own directory somewhere in the `pkgs/` tree, i.e. in `pkgs/`*`category`*`/`*`subcategory`*`/.../`*`pkgname`*. Below are some rules for picking the right category for a package. Many packages fall under several categories; what matters is the *primary* purpose of a package. For example, the `libxml2` package builds both a library and some tools; but it's a library foremost, so it goes under `pkgs/development/libraries`.

When in doubt, consider refactoring the `pkgs/` tree, e.g. creating new categories or splitting up an existing category.

If it's used to support *software development*:

- **If it's a *library* used by other packages:** `development/libraries` (e.g. `libxml2`)
- **If it's a *compiler*:** `development/compilers` (e.g. `gcc`)
- **If it's an *interpreter*:** `development/interpreters` (e.g. `guile`)
- **If it's a (set of) development *tool(s)*:**
  - **If it's a *parser generator* (including lexers):** `development/tools/parsing` (e.g. `bison`, `flex`)
  - **If it's a *build manager*:** `development/tools/build-managers` (e.g. `gnumake`)
  - **Else:** `development/tools/misc` (e.g. `binutils`)
- **Else:** `development/misc`

If it's a (set of) *tool(s)*: (A tool is a relatively small program, especially one intented to be used non-interactively.)

- **If it's for *networking*:** `tools/networking` (e.g. `wget`)
- **If it's for *text processing*:** `tools/text` (e.g. `diffutils`)
- **If it's a *system utility*, i.e., something related or essential to the operation of a system:** `tools/system` (e.g. `cron`)
- **If it's an *archiver* (which may include a compression function):** `tools/archivers` (e.g. `zip`, `tar`)
- **If it's a *compression* program:** `tools/compression` (e.g. `gzip`, `bzip2`)
- **If it's a *security*-related program:** `tools/security` (e.g. `nmap`, `gnupg`)
- **Else:** `tools/misc`

If it's a *shell*: `shells` (e.g. `bash`)

If it's a *server*:

- **If it's a web server:** `servers/http` (e.g. `apache-httpd`)
- **If it's an implementation of the X Windowing System:** `servers/x11` (e.g. `xorg` — this includes the client libraries and programs)
- **Else:** `servers/misc`

If it's a *desktop environment* (including *window managers*): `desktops` (e.g. `kde`, `gnome`, `enlightenment`)

If it's an *application*: A (typically large) program with a distinct user interface, primarily used interactively.

- **If it's a *version management system*:** `applications/version-management` (e.g. `subversion`)
- **If it's for *video playback / editing*:** `applications/video` (e.g. `vlc`)
- **If it's for *graphics viewing / editing*:** `applications/graphics` (e.g. `gimp`)
- **If it's for *networking*:**
  - **If it's a *mailreader*:** `applications/networking/mailreaders` (e.g. `thunderbird`)

If it's a *newsreader*: `applications/networking/newsreaders` (e.g. `pan`)
If it's a *web browser*: `applications/networking/browsers` (e.g. `firefox`)
Else: `applications/networking/misc`

Else: `applications/misc`

If it's *data* (i.e., does not have a straight-forward executable semantics):

If it's a *font*: `data/fonts`

If it's related to *SGML/XML processing*:

If it's an *XML DTD*: `data/sgml+xml/schemas/xml-dtd` (e.g. `docbook`)
If it's an *XSLT stylesheet*: (Okay, these are executable...)
`data/sgml+xml/stylesheets/xslt` (e.g. `docbook-xsl`)

If it's a *game*: `games`

Else: `misc`

7.3.2 Versioning

Because every version of a package in Nixpkgs creates a potential maintenance burden, old versions of a package should not be kept unless there is a good reason to do so. For instance, Nixpkgs contains several versions of GCC because other packages don't build with the latest version of GCC. Other examples are having both the latest stable and latest pre-release version of a package, or to keep several major releases of an application that differ significantly in functionality.

If there is only one version of a package, its Nix expression should be named `e2fsprogs/default.nix`. If there are multiple versions, this should be reflected in the filename, e.g. `e2fsprogs/1.41.8.nix` and `e2fsprogs/1.41.9.nix`. The version in the filename should leave out unnecessary detail. For instance, if we keep the latest Firefox 2.0.x and 3.5.x versions in Nixpkgs, they should be named `firefox/2.0.nix` and `firefox/3.5.nix`, respectively (which, at a given point, might contain versions `2.0.0.20` and `3.5.4`). If a version requires many auxiliary files, you can use a subdirectory for each version, e.g. `firefox/2.0/default.nix` and `firefox/3.5/default.nix`.

All versions of a package *must* be included in `all-packages.nix` to make sure that they evaluate correctly.