elba Documentation

Release 0.3.0

David Cao

Mar 19, 2019

Usage

elba is a package manager for the Idris programming language. This book aims to be a mostly comprehensive guide on actually using it.

CHAPTER 1

Getting Started

This section is for getting up-to-speed with elba as fast as possible, covering getting elba installed on your machine in the first place and making a new project.

By the end of this chapter, you should have a basic elba installation up-and-running, as well as a general overview of how to use elba for day-to-day Idris development.

1.1 Installation

The easiest and most convenient way of installing elba is to use the pre-built binaries for elba, which can be downloaded from GitHub Releases. To install this way, just download the corresponding archive for your platform, extract the executable somewhere in your PATH, add `~/.elba/bin` to your PATH in order to execute elba-installed packages, and you're done!

Note: For Linux platforms, there are two varieties of binaries available: one suffixed with `-gnu` and the other suffixed with `-musl`. The `-gnu` binary is dynamically linked to the system libc, while the `-musl` binary is statically linked using musl.

For most users, the `-gnu` binary should work fine, but if it doesn't, try using the `-musl` binary.

1.1.1 Installing with Cargo

Because elba is written in Rust, it is available as an installable crate from crates.io. In order to install elba this way, you should have a copy of the Rust toolchain installed on your computer first. The process for this is explained on the Rust website. The minimum stable version of Rust elba has successfully been built on is **1.31**.

Once you have Rust installed, installing elba is self-explanatory:

```
$ cargo install elba
$ elba # should work
```

Remember to add `~/.elba/bin` to your PATH to be able to run elba-installed packages.

1.1.2 Building elba

Building elba from source is much the same process as installing it using cargo; the only difference is that instead of using a stable, versioned-crate available from crates.io, elba's source code is used directly. You'll still need to have Rust **1.31** or later installed. After that's done, download elba's source code and install it:

```
$ git clone https://github.com/elba/elba
$ cd elba
$ cargo install --release
$ elba # should work!
```

Remember to add `~/.elba/bin` to your PATH to be able to run elba-installed packages.

1.2 Quick Start

This section intends to be a whirlwind tour of all the functionality available with elba. For more information on each step, refer to either the Usage or Reference chapters.

1.2.1 Creating a package

Creating a package is easy with elba: all you need is a package name. Note that names in elba are special in that they are *always namespaced*; every name in elba comes with a group part and a name part, separated with a slash. For more information, see the information on names in the *manifest chapter*.

```
$ elba new asd # won't work: no namespace
$ elba new grp/asd # ok!
```

This command will generate a new elba project with name `grp/asd` in the folder `./asd/`, along with an associated git project. If you want to omit the git project, pass the option `--vcs none`.

By default, elba will create a project with a binary target, with a main file located at `src/Main.idr`. If you'd like to generate a package with a library target instead, pass the `--lib` flag, which will add a library target to the manifest and generate the file `src/{group}/{name}.idr`. This file structure of having a group followed by a name is just convention, and isn't required.

Regardless of which target is chosen, an `elba.toml` manifest file will also be generated.

Initializing a pre-existing package

If you already have an Idris project and want to turn it into an elba project, use the `elba init` command instead; it follows the exact same syntax as `elba new` and is functionally identical, but uses the current directory instead of making a new one.

1.2.2 Adding dependencies

Now that a new package has been created, you can start to add packages as part of your dependencies. A package can originate from one of three places: a git repository, a file directory, or a package index. Ordinary dependencies are placed under the `[dependencies]` section, while dependencies that are only needed for tests and the like are placed under `[dev_dependencies]`. Examples are shown below:

```
[dependencies]
"index/version" = "0.1.5" # uses the default index (i.e. the first specified one in␣
↪configuration)
"index/explicit" = { version = "0.1.5", index = "index+dir+../index" } # uses the␣
↪index specified
"directory/only" = { path = "../awesome" } # uses the package in the path specified
"git/master" = { git = "https://github.com/doesnt/exist" } # uses the master branch
"git/explicit" = { git = "https://github.com/doesnt/exist", branch = "beta" } #
↪"branch" can be an arbitrary git ref: a tag, commit, etc.
```

For more information on the syntax regarding specifying and adding custom indices, see the chapters on *Resolutions* and *Configuration*. More information about dependency specification syntax is available at *its relevant chapter*.

Note that only packages with library targets can be depended on.

At this point, you can add whatever files you want and import anything from your dependencies.

1.2.3 Targets

The manifest also allows you to specify which targets you want to have built for your package. There are three types of targets:

- A **library target** allows this package to be depended on by other packages. A package can only have one library, and the syntax follows the following:

```
[targets.lib]
# the path which contains all of the lib files (*cannot* be a parent directory)
# this is set to "src" by default
path = "src/"
# a list of files to export
mods = [
    "Awesome.A", # the file src/Awesome/A.idr
    "Control.Zygohistomorphic.Prepromorphisms", # the file src/Control/
↪Zygohistomorphic/Prepromorphisms.idr
]
```

- A **bin target** specifies a binary to be built. Multiple binaries can correspond to one package.

```
[[targets.bin]]
# the name of the binary to create
name = "awes"
# the path which contains all of the bin files (*cannot* be a parent directory)
# this is set to "src" by default
path = "src/"
# the path to the Main module of the binary
main = "Awesome.B"
```

  Note: the format of the binary target has some nuance to it, so for more information, see the docs on *the manifest format*.

- A **test target** specifies a test binary to build. It uses the same syntax as a bin target, with the difference that we use `[[targets.test]]` to specify them and the test binary can depend on the dev-dependencies as well as the root package's library. A test binary succeeds upon execution if it returns exit code 0.

1.2.4 Building a package

. . . can be accomplished with the command:

```
$ # assuming the current directory is an elba package
$ elba build
```

For all elba build-related commands, the `IDRIS_OPTS` environment variable will dictate additional arguments to pass to the Idris compiler (the flags passed by elba get higher priority). Additionally, any args passed after a double-dash will be interpreted as arguments to the Idris compiler:

```
$ # adds both the contrib and effects built-in packages
$ IDRIS_OPTS="-p contrib" elba build -- -p effects
```

When building a local package, the output binaries are located at `target/bin`, while the output library is placed at `target/lib`.

Interactive development with the REPL can also be accomplished with the command:

```
$ # assuming the current directory is an elba package
$ elba repl
```

Instead of placing the build outputs in a `target/` folder, the `elba repl` command directly loads the files on-disk, then cleans up any build files after execution.

elba uses an `elba.lock` lockfile to ensure that these builds are reproducible. This should be committed to repositories for libraries, but not for binaries.

CHAPTER 2

Configuration

elba's behavior can be configured through the use of TOML configuration files and environment variables. elba checks the current directory and all of its ancestors for a `.elba/config` file, unifying them in the following order (from highest to lowest priority):

```
# assuming current directory is /foo/bar/baz/quux
/foo/bar/baz/quux/.elba/config
/foo/bar/baz/.elba/config
/foo/bar/.elba/config
/foo/.elba/config
/.elba/config
# Your platform-specific config file would go here
# - Linux: ~/.config/elba/config
# - macOS: /Users/<user>/Library/Preferences/elba/config
# - Windows: %LOCALAPPDATA%\elba\config\config
$HOME/.elba/config
```

Any specified environment variables have the highest priority. This behavior heavily borrows from Cargo's configuration format.

Additionally, whenever elba executes an Idris invocation, elba will pass all of the arguments in the environment variable `IDRIS_OPTS` to the compiler. In any case where the `IDRIS_OPTS` args conflict with elba's own flags (i.e. if the user specifies the flag `--ide-mode` but elba specifies `--check`), elba will override the user-specified flag.

2.1 Config Format

A complete default elba configuration file is listed below. Any options which are not assigned to will carry the default value instead.

```
compiler = "idris"

[indices]
"official" = "index+git+https://github.com/elba/elba"
```

(continues on next page)

(continued from previous page)

```
[term]
verbosity = "normal"
color = "true"

[alias]
i = "install"
b = "build"
t = "test"

[directories]
cache = "$HOME/.elba"

[[backend]]
name = "c"
default = true
portable = false
opts = []
```

Using environment variables

In order to specify an option as an environment variable, simply replace the "dots" of the option with underscores, and prefix with `ELBA_`. So the option `term.verbosity` becomes `ELBA_TERM_VERBOSITY`.

2.1.1 `compiler`

The compiler key specifies the name of the executable of the Idris compiler. By default it is set to "idris". You should **not** pass any command line options in this string, as elba will search the path for an executable with the name of this string.

elba is smart enough to detect the version of the compiler - whether it's Idris 1 or 2 (Blodwen). If it can't tell what version the compiler is, it'll default to the behavior for Idris 1.

2.1.2 `indices`

This key plays a few different roles based on the context of the elba operation:

- When building a local package or running a command which takes a `--index` command-line flag, this key defines aliases for indices; this way, you don't have to completely write out the resolution of an index to refer to it (but you can if you want).

  For both the command-line flag and when building a package, if an index is specified, elba will first see if it's an alias for another index. If not, it will try to parse the index as an index resolution.

  The first index specified is set as the default index when building a package. For commands with an `--index` flag, elba will require that you specify what index you're referring to if the config lists multiple indices.

- When building a package which originates from an index, this key defines all the indices that will be searched for the package.

By default, the first and only index available to elba is the official package index.

2.1.3 [profile]

This section specifies the default author information that should be provided upon creating or initializing a new elba project. By default, this section has no value, so new projects are made without an author.

```
[profile]
name = "John Smith"
email = "jsmith@example.com"
```

2.1.4 [term]

This section specifies options for terminal output, and has two fields:

- `verbosity`: specifies how verbose elba should be. Can be one of `verbose`, `normal`, `quiet`, or `none`.
- `color`: specifies if elba should try to print color output. Either `true` or `false`.

At the moment, neither of these options actually do anything.

2.1.5 [alias]

This section is for providing aliases for commands. The key represents the alias and the value represents the the command that it should be aliased to. Note that aliases can alias to other aliases, which can cause *infinite recursion of aliases*. Be careful.

```
$ elba b # builds the local package with the default alias settings
```

2.1.6 [directories]

This section only contains one key: `cache`, for the location where the global cache should be placed. This controls not only the location of elba's temporary build directories but also the location of the global bin directory.

2.1.7 [[backend]]

This section specifies information about codegen backends. By default, information about one default codegen is provided: the C backend. These settings are used whenever a codegen backend is unspecified or a codegen backend is specified but doesn't have any information on it available in the configuration. A example full `[[backend]]` section is provided below:

```
[[backend]]
# The name of the backend, passed to the --codegen or --portable-codegen
# compiler option
name = "awesome"
# Whether this should be treated as a new default codegen backend, instead of
# the c one provided by default. Note that if multiple backends have default set
# to true, the backend mentioned first will be used as the default
default = true
# Whether or not this backend is portable
portable = false
# The command to use to run executables generated by this codegen backend
# If omitted, the executable will just be run by itself
runner = "awesomec"
```

(continues on next page)

(continued from previous page)

```
# The extension to use for executables generated by this codegen backend
# elba will pass the name of the binary/test target with this extension set to
# the -o flag of the Idris compiler
# If unset, no extension-setting will happen
extension = "awe"
# Options to be passed to the codegen backend
opts = []
```

CHAPTER 3

Installing a Package

elba can build and install the binary targets of packages into a global directory (this directory is the `bin` subfolder under the folder of the global cache; under normal circumstances, this should be located at `~/.elba/bin`). In order for these executables to be run from anywhere, you should this global bin folder to your `PATH`.

3.1 Installing a local package

To install a package which is located on-disk, simply navigate to the directory of the package and whack:

```
$ elba install
```

Doing that should rebuild the package if needed and install its binaries into the global bin folder.

Note that if a binary with the same name as one of the binaries being installed already exists, the above command will fail. If you're absolutely sure that you want to replace the old binary, run the command again but with the `--force` flag. Additionally, if you only want to install certain binaries, you can use the `--bin` flag:

```
$ elba install --bin yeet # only install the binary named "yeet"
```

3.2 Installing a package from an index

If one or more package indices is *specified in elba's configuration*, you also have the option of installing a package from one of those indices. `elba install` optionally takes a **package spec** as an argument, which consists of three parts:

- The name of the package to install (required)
- The **resolution** of the package; for the time being, this must be the resolution of an index (see *Resolutions*)
- The version of the package

The following are examples of valid `elba install` invocations:

```
$ # installs the latest version of `jsmith/one` from any index it can:
$ elba install "jsmith/one"
$ # installs version 1.0.0 of `jsmith/one` from any index it can:
$ elba install "jsmith/one|1.0.0"
$ # installs the latest version of `jsmith/one` from the index specified:
$ elba install "jsmith/one@index+tar+https://example.com/index.tar.gz"
$ # installs version 1.0.0 of `jsmith/one` from the index specified:
$ elba install "jsmith/one@index+tar+https://example.com/index.tar.gz|1.0.0"
```

As with installing a local package, if you want to replace any old binaries in the global bin directory, use the `--force` flag, and if you want to choose which binaries to install, use the `--bin` flag.

Note that if a spec can apply to multiple packages at the same time (i.e. a package index wasn't specified and multiple package indices offer a package with the same name), elba will require you to provide more info to disambiguate between the packages.

3.3 Uninstalling a package

Uninstalling a package is much the same process as installing: just pass a spec to the `elba uninstall` invocation. Just like with `elba install`, if you specify an ambiguous spec, elba will require you to qualify it further.

CHAPTER 4

Custom Subcommands

To support extensibility in the future, elba supports running custom subcommands if it is passed a subcommand which doesn't exist. All arguments which were passed to elba will be instead passed to the subcommand:

```
$ elba installnt # executes `elba-installnt`
$ elba installnt awesome one two three # executes `elba-installnt awesome one two␣
↪three`
$ elba installnt --cool awesome --one -f # executes `elba-installnt --cool awesome --
↪one -f`
```

elba is also available as a Rust library, meaning that subcommands written in Rust can take advantage of elba's internal data structures and functions. This opens a variety of possibilities: using custom project scaffolds and templates, running special heuristics on elba projects, etc.

CHAPTER 5

Publishing to an Index

TODO: Rewriting for 0.3.0...

CHAPTER 6

Resolutions

A core tenet in elba's functionality is the idea of **resolutions**. A resolution is a generic location from which some resource (a package or a *package index*) can be retrieved. Internally, elba distinguishes between two types of resolutions:

- A **direct resolution** refers to a direct location from which a resource (either a package or a package index) can be downloaded. Direct resolutions themselves can include references to tarballs (either on a network somewhere or located on disk), local directories on disk, or git repositories.
- An **index resolution** refers to an index from which information about a package's location can be obtained. The location of the index itself must be a direct resolution.

A package can have (and is identified by) either a direct resolution or an index resolution. A package index is identified by its index resolution.

6.1 Syntax

In order to refer to these types of direct resolutions, elba has its own simple syntax for "resolution strings":

- Each of the types of direct resolutions has its own syntax:
  - For a direct resolution which points to a tarball, the resolution string must start with the identifier `tar+` and include a properly-formed URL with either the `http://`/`https://` (referring to a tarball on the network somewhere) or `file://` (referring to a local tarball) schemas:

```
These are all valid:
tar+http://example.com/asdf.tar.gz
tar+https://example.com/asdf
tar+file://../asdf.tar.gz
```

  - For a direct resolution which points to a directory on disk, the resolution string must start with the identifier `dir+` and include a properly-formed path to a directory on disk:

```
These are all valid:
dir+asdf
```

(continues on next page)

(continued from previous page)

```
dir+./asdf
dir+../asdf/whatever/subfolder

On Windows, these would be valid too:
dir+C:\Users\John\etc
```

  - For a direct resolution which points to a git repository, the resolution string must start with the identifier `git+` and provide the URL of the repository in question. Additionally, a git ref can be specified as part of the fragment of the URL:

```
These are all valid:
git+https://github.com/example/doesnt-exist
git+https://github.com/example/doesnt-exist#master <- use the master branch
git+https://github.com/example/doesnt-exist#v1.0.0 <- use the "v1.0.0" tag
git+https://github.com/example/doesnt-exist#a4e13343 <- use the commit
↪"a4e13343"
git+ssh://git@github.com/example/doesnt-exist <- using ssh instead of https
```

- For an index resolution, the resolution string must start with the identifier `index+` and include the direct resolution of the origin of the index:

```
These are all valid
index+tar+http://example.com/asdf.tar.gz
index+dir+../asdf/whatever/subfolder
index+git+ssh://git@github.com/example/doesnt-exist#a4e13343
```

CHAPTER 7

The Manifest

In order to keep track of package metadata like the name of a package and what targets should be built for that particular package, elba uses an `elba.toml` manifest file. This file is divided into several different sections which each provide information to elba about the package in question.

7.1 [package]

The first and most important section of the manifest is the `[package]` section, which lists all of the metadata about the package. A complete example of a `[package]` section is shown below:

```
[package]
name = "dcao/elba"
version = "0.1.0"
authors = ["David Cao <dcao@example.com>"]
description = "The best package ever released"
homepage = "https://github.com/elba/elba"
repository = "https://github.com/elba/elba"
readme = "README.md"
license = "MIT"
keywords = ["package-manager", "packaging"]
exclude = ["*.blah"]
```

The namespaced name and version are the two most important parts of this specification. The name must contain a group (i.e. a namespace) and a name, separated by a slash, or else the manifest will fail to parse. Additionally, the name can only contain alphanumeric characters, hyphens, and underscores. Internally, elba ignores case and treats hyphens and underscores equally when deciding if two names are identical. The version must follow Semantic Version guidelines. Additionally, the package section contains fields to indicate the authors of the package and the license which the code falls under. The authors section can be left empty, and each author should follow the format `name <email>` (this is just a helpful convention to follow). The license field can be omitted entirely, as can the description, homepage, repository, readme, and keywords.

Note: Why namespacing?

Having to supply a namespace to all package names might seem like unnecessary work, but it has its benefits; this design decision to require all package names to be namespaced was borne out of observations of other package ecosystems where the lack of namespaces lead to significant problems down the line. In particular, namespaced packages provide the following benefits:

- Packages which should belong to a single "group" or are a part of a single ecosystem can easily be grouped together, rather than using ad-hoc kinda-sorta-namespacing by prefixing all related packages with some name, which any untrusted package uploader can do
- Name-squatting becomes less of an issue; instead of one global `http` package in a package index, there are now separate `jsmith/http` or `whatever/http` packages
- Namespacing doesn't stop people from coming up with "creative" names; you can still create a package called `jsmith/unicorns_and_butterflies` if you'd like.

The `exclude` field specifies files which should be ignored when building a package, packaging a package, and checking to see if the package has changed. Each element of the list should correspond to a line in a `.gitignore` file. Also note that if an actual `.gitignore` file is present, elba will also ignore any files as specified by that file.

7.2 `[dependencies]` and `[dev_dependencies]`

These sections of the manifest are mostly self-explanatory; they're a place where you can specify the dependencies that your package needs. All packages in the `[dependencies]` section will be loaded for every target of the package, while the packages in the `[dev_dependencies]` section will only be loaded for test targets.

elba dependencies can originate from one of three places: a package index (think RubyGems or crates.io), in which the package is identified by its version and package index (defaulting to the first package index specified in the *config file*; a git repository, in which the package is identified by the url of the git repo and a git ref name (defaulting to "master"); and a directory tree, in which the package is identified by its path.

An example of these sections and all the types of dependencies is shown below:

```
# deps used for all targets
[dependencies]
"index/version" = "0.1.5" # uses the default index (i.e. the first specified one in␣
↪configuration)
"index/explicit" = { version = "0.1.5", index = "index+dir+../index" } # uses the␣
↪index specified
"directory/only" = { path = "../awesome" } # uses the package in the path specified

# deps only used for the test targets
[dev_dependencies]
"git/master" = { git = "https://github.com/doesnt/exist" } # uses the master branch
"git/explicit" = { git = "https://github.com/doesnt/exist", tag = "beta" } # "tag"␣
↪can be an arbitrary git ref: a tag, commit, etc.
```

elba's syntax for versioning has *several idiosyncrasies of its own*, but the tl;dr version is that elba will always pick a version of that package which is greater than or equal to and semver compatible with the version specified.

For more information about package indices, see the *relevant reference page*.

7.3 `[targets]`

In order to know which files to build and how to build them, elba manifest files also must specify a `[targets]` section. There are three types of targets which elba can build:

- A **library target** is exactly what it sounds like: a built library of ibc files which can be used and imported by other elba packages. Each package can only export a single library target; attempting to specify multiple library targets will result in a manifest parsing error. The syntax for a library target is as follows:

```
[targets.lib]
# The path to the library - defaults to "src"
path = "src"
# The list of files which should be exported and made available for public use
mods = [
  "Awesome.A", # the file src/Awesome/A.idr, or src/Awesome/A.lidr
  "Control.Zygohistomorphic.Prepromorphisms", # the file src/Control/
↪Zygohistomorphic/Prepromorphisms.idr,
                                              # or src/Control/Zygohistomorphic/
↪Prepromorphisms.lidr
]
# Optional flags to pass to the compiler
idris_opts = ["--warnpartial", "-p", "effects"]
```

  The `path` key should be a **sub-path** of the package; it cannot reference parent or absolute directories of the package. During the build process, all of the files under the `path` sub-path will be used to build the library and export the Idris bytecode files corresponding to the items in `mods`.

- A **binary target** is a binary which should be generated based on a Main module. Packages can have as many binary targets as they please; by default, all binary targets are built/installed in an `elba build` or `elba install` invocation, but this can be changed with the `--bin` flag. The syntax for a binary target is as follows:

```
[[targets.bin]]
# The name of the output binary
name = "whatever"
# The path to the folder containing the binary source - defaults to "src"
path = "src/bin"
# The path to the Main module
main = "Whatever" # corresponds to src/bin/Whatever.idr
# Optional flags to pass to the compiler
idris_opts = ["--warnpartial", "-p", "effects"]
```

  The `name`, and `idris_opts` fields should be self-explanatory, but the `path` and `main` arguments have some more nuance to them. In order to maintain backwards compatibility while providing maximum flexibility, elba follows several steps to resolve the location of a binary target. It's pretty hard to explain these steps, but examples are much easier to follow:

```
# Example 1: strict subpath specified in main, with folders separated by
# slashes. extension left unspecified.
main = "bin/Whatever/Module"
# this corresponds to the first of the following files which exists:
# - bin/Whatever/Module.idr
# - bin/Whatever/Module.lidr
# - src/bin/Whatever/Module.idr (because of the default `path` value)

# Example 2: main uses dots instead of slashes to separate folders, and
# includes an idr extension
main = "Whatever.Module.idr"
# because this is not a valid subpath (uses dots instead of slashes),
# this corresponds to the first of the following files which exists:
# - src/Whatever/Module/idr.idr (treat the last section as a module)
# - src/Whatever/Module/idr.lidr (same, but literate file)
# - src/Whatever/Module.idr (treat the last section as an extension:
#                            applies to the "idr" extension only)
```

(continues on next page)

(continued from previous page)

```
# - src/Whatever/Module.lidr (same, but literate file)
# this file should have a function Main.main

# Example 3: strict subpath specified with non-"idr" extension
main = "bin/Whatever/Module.custom"
# corresponds to the first of the following files which exists:
# - bin/Whatever/Module.idr
# - bin/Whatever/Module.lidr
# - src/bin/Whatever/Module.idr (due to the default `path` value)
# - src/bin/Whatever/Module.lidr
# in both cases, this file should have a function `Module.custom : IO ()`,
# which will be used as the main function

# Example 4: non-subpath combined with custom path and non-"idr" extension
path = "bin"
main = "Whatever.Module.custom"
# corresponds to the first of the following files which exists:
# - bin/Whatever/Module/custom.idr (treat the last section as a module)
# - bin/Whatever/Module/custom.lidr
# - bin/Whatever/Module.idr (treat the last section as a function in a parent␣
↪module)
# - bin/Whatever/Module.lidr
# if this corresponds to `bin/Whatever/Module.idr`, then the file should have a
# function `Whatever.Module.custom : IO ()`, which will be used as the main
# function
```

- A **test target** shares many similarities with a binary target: the syntax is almost exactly the same, and a single package can have multiple test targets. Indeed, in elba, tests are just executables which return **exit code 0 on success** and **any other exit code on failure**. The distinguishing features of a test target are as follows:
  - The `path` value for test targets defaults to `tests/` instead of `src/`
  - The `name` value defaults to the value in main, with slashes and periods replaced with underscores and `test-` prepended.
  - Test targets have access to (i.e. can import from) **all dev dependencies** along with **the package's own library target**.

    This means that if you want to test a library target, you don't have to do anything special, just import your library like you normally would.

    If you want to test a binary, you can still do this, since a test will be built with all of the files in the same directory as the test's Main module, so if you put your test's Main module in the folder as a binary target, you can import everything that your binary target can from within the test.
  - Test targets can be automatically built and run in one shot using the command `elba test`.

  You'll note that the syntax for specifying a test target is remarkably similar to that for specifying a binary target:

```
# The name of the output test binary
name = "test-a"
# The path to the test's Main module
main = "tests/TestA.idr"
# Optional flags to pass to the compiler
idris_opts = ["--warnpartial"]
```

An elba package **must** specify either a lib target or a bin target, or else the manifest will be rejected as invalid.

For local packages, after building, all binaries will be output to the `target/bin` folder, and any library will be output

to the `target/lib` folder. Additionally, for libraries, if you pass the `--lib-cg` flag, elba will use the codegen backend specified (or the C backend by default) and any export lists specified in the exported files of the library to create output files under `target/artifacts/<codegen name>` (for more information on export lists and the like, see this test case in the Idris compiler).

7.3.1 Virtual packages

elba allows packages to declare no packages at all; packages without any targets are called **virtual packages**.

7.4 [scripts]

elba can run arbitrary shell commands called **scripts.** These are defined in a package's manifest file under the `[scripts]` section:

```
[scripts]
"prebuild" = "echo 'I'm building now!"
"whatever" = "echo 'Hey!'"
"dep" = "elba script whatever && echo 'Cool.'"
```

These can manually be executed with the *elba script* subcommand:

```
$ elba script whatever
```

This feature is deceptively simple; because scripts can call other scripts in the same project, these simple scripts can function as a viable alternative to task runners like `make`.

Additionally, elba has a concept of **hooks**, which are scripts that are automatically run during certain phases of the build and install process. Currently, there is only one hook: `prebuild`, which, if defined, is run automatically right before a package is built.

7.5 [workspace]

The last section in the manifest is the workspace section, used to indicate subprojects in the current directory. At the moment, the only use for this field is to indicate to elba the location of a package in a subdirectory (for example, with if a git repo has a package located in some subdirectory). Adding a package to the local workspace *does not* automatically add it as a local dependency of the package, nor does it cause the workspace packages to be automatically built when the root package is built. To add local directories as dependencies, they must manually be specified in either the `[dependencies]` or `[dev_dependencies]` sections.

Note that the directory of every package must be a **sub-path**; it cannot refer to an absolute directory or a directory above the root package.

An example workspace section is shown below:

```
[workspace]
"name/one" = "pkgs/one"
"other/pkg" = "wherever/youd/like"
```

Note that a a `[workspace]` section can stand alone and be parsed as a valid manifest if there is no package in the root directory.

7.6 An aside: the lockfile

In order to keep track of the dependency tree and create reproducible builds, elba uses a lockfile called `elba.lock`. This lockfile **should not be modified** in any way, as it can lead to unpredictable results during the build process.

The lockfile will not change so long as all of the packages in the lockfile satisfy the requirements of the manifest and of its transitive dependencies. For git repositories, the lockfile will lock a package to a commit, which won't change given that the following conditions hold:

- If the manifest references a branch, the locked commit must be contained within that branch.
- If the manifest references a specific tag or commit, the locked commit must be equal to that tag or commit.

CHAPTER 8

Indices

A **package index** is a source of metadata for available packages, mapping package names and versions to requisite dependencies and a location to retrieve the package. Package indices serve several purposes in elba's package management system:

- Package indices group together versions of packages to make depending on and installing packages easier, more convenient, and less prone to breakage (á la RubyGems, crates.io)
- Package indices can serve to curate sets of packages which are known to work together correctly (á la Stackage)
- They provide a level of indirection for packages; consumers of packages don't have to be tied to directly depending on a certain git repository or tarball, they can just rely on wherever the index says the package is located.

Packages within package indices are capable of depending on packages in other indices (so long as the index specifies all of the indices it depends on), and users of elba can specify multiple package indices to pull from. Additionally, packages in package indices can have arbitrary direct resolutions as their actual location. This makes elba's package indices extremely powerful as a consequence.

Users can have their packages appear in indices by uploading them to *index backends*.

8.1 Index Resolutions

An index is identified primarily by its index resolution, which corresponds to the place where the index is made available. For more information, see the previous chapter on *Resolutions*.

In the `elba.toml` file, when a package requirement is declared with a certain version, elba goes through the following steps to decide which package index to use:

- If the resolution of an index is provided in the dependency specification, elba will use that index.

```
[dependencies]
"test/one" = { version = "0.1.0", index = "index+dir+/index" }
# for this package, elba will use the index located on-disk at `/index`.
```

- If no resolution is provided, elba will default to *the first index listed in configuration*.

```
# .elba/config
indices = [
    "index+dir+/one",
    "index+dir+/two"
]

# elba.toml
[dependencies]
"test/two" = "0.1.0"
# for this package, elba will use the index located on-disk at `/one`.
```

Note that if a declared dependency uses an index that isn't specified in the configuration, the package will fail to build during dependency resolution with a "package not found" error.

8.2 `index.toml`

A package index is (when extracted, for tarballs) a directory tree of metadata files. All package indices must have a configuration file at the root of this directory tree named `index.toml`, and specify the following keys:

```
[index]
secure = false

[index.dependencies]
```

The `secure` key tells elba whether to treat the index like a secure package index. At the moment, this flag does nothing, but in the future, this flag may be used to enable compatibility with The Update Framework. For forwards compatibility, package index maintainers should set this key to `false`.

The `dependencies` key is a mapping from the "name" of an index to its index resolution. The name can be whatever you want, but that name will be how the index will be referred to within metadata files. Every other index which the packages of this index need to build properly must be specified in this field, or else package building will fail during dependency resolution.

An additional key, `backend`, should be the url of the backend API.

8.3 Metadata structure

Package indices must follow a fairly strict folder and file structure in order for elba to interpret them correctly. The top-level folders should be groups, and underneath the folder for each group should be a metadata file corresponding to a package. The name of that file should be the second portion of the package's name:

```
# an example index:
.
|-- group
|   |-- name # metadata file corresponding to the package `group/name`
|   +-- cool # metadata file corresponding to the package `group/cool`
|-- next
|   +-- zzz # metadata file corresponding to the package `next/zzz`
|
+-- index.toml
```

Each line of the metadata file for a package should be a complete JSON object corresponding to a specific version of a package, and should follow the following structure (pretty-printed for readability):

```
{
  "name": "no_conflict/root",
  "version": "1.0.0",
  "dependencies": [
    {
      "name": "no_conflict/foo",
      "req": "1.0.0"
    },
    {
      "name": "awesome/bar",
      "index": "best_index",
      "req": ">= 0.1.0"
    }
  ],
  "yanked": false,
  "location": "dir+test"
}
```

The `name` and `version` fields should be self-explanatory. The `dependencies` section should be a list of objects with fields `name`, `index`, and `req`. `name` is self-explanatory, and `req` is just the version constraint of that particular dependency. The value in `index` should correspond to an index name specified within the index's config; if the index is unspecified or if the index name can't be found in configuration, elba will assume that the package is available from the current index.

The `yanked` field allows for "yanking" of a package, which disallows future consumers of a package from using that version (but allows current consumers of a yanked package version to continue using it). Finally, the `location` field indicates the direct resolution of the package in question.

8.4 Index Retrieval Semantics

To avoid constantly updating the package index, elba will only update its indices if it's building a global project (i.e. `elba install`), or if a package cannot be found in the locally cached indices or changes versions in such a way that is incompatible with an existing lockfile. This means that if an index changes the resolution of a package, the package indices might not be updated immediately.

CHAPTER 9

Index Backends

TODO...

CHAPTER 10

Dependencies

The most important job of a package manager is building dependencies of a package. Packages in elba can depend on other packages in external *indices*, a local file directory, or a git repository.

10.1 Versions

Versions in elba follow a slightly modified version of Semantic Versioning in order to ensure that packages stay compatible with each other. Most of the core concepts of Semantic Versioning are carried over:

- Differences in the major version indicate backwards incompatibility.
- Differences in the minor version indicate feature additions.
- Differences in the patch version indicate bug fixes or other non-feature additions.
- Pre-release versions can be indicated with suffixes: `1.0.0-pre.2-beta.5`

In version constraints, the second and third components of a version can be omitted, in which case they are assumed to be `0`. A pre-release cannot be specified without also specifying the second and third components.

10.1.1 Version constraints

We say that a constraint **satisfies** a particular version if that particular version falls within the version constraint.

elba's version constraints offer all the same standard operators (<, >, ^, ~, etc.), but they have some idiosyncrasies which distinguish them from how other package managers work.

10.1.2 Inequality constraints

The "lowest-level" constraints elba offers are **inequality constraints**, which are fairly simple: `< 1.0.0`, `>= 1.0.0`, etc.

By default, **"<" constraints will ignore pre-release versions**. for ergonomic reasons. If a package specifies that they depend on `< 1.0.0`, they likely don't want to have any of the pre-release versions of 1.0.0 selected, even if those technically satisfy the constraint. If a package wants to include the pre-release versions as well it can opt in to pre-releases by adding a bang after the constraint symbol like so: `<! 1.0.0`.

The bang trick also works for >= constraints as well: while `>= 1.0.0` doesn't match pre-releases of `1.0.0`, `>=! 1.0.0` does.

The constraint parser will allow you to add bangs to all types of less-than or greater-than constraints, but some of them won't do anything: `<= 1.0.0` and `<=! 1.0.0` mean the exact same thing, as do `> 1.0.0` and `>! 1.0.0`.

Additionally, if the constraint specifies a pre-release, it will satisfy other pre-releases.

Two inequality constraints can be intersected to produce a new compound constraint. Note that at the moment, this is the only case in which the parser will accept multiple constraints. Additionally, the greater-than bound must be written before the less-than bound.

The new constraint must allow at least one version for it to be valid:

```
>= 1.0.0 < 1.4.2 # valid
>= 1.0.0 <= 1.0.0 # valid
< 1 > 0 # invalid: less-than specified before greater-than
> 1 < 0 # invalid: impossible constraint (satisfies no versions)
```

10.1.3 Caret constraints

Caret constraints in elba function the same as in other package managers. To quote Cargo's documentation:

> Caret requirements allow SemVer compatible updates to a specified version. An update is allowed if the new version number does not modify the left-most non-zero digit in the major, minor, patch grouping.

Here are some examples of caret constraints (also taken from Cargo's documentation):

```
^1.2.3 := >= 1.2.3 < 2.0.0
^1.2   := >= 1.2.0 < 2.0.0
^1     := >= 1.0.0 < 2.0.0
^0.2.3 := >= 0.2.3 < 0.3.0
^0.2   := >= 0.2.0 < 0.3.0
^0.0.3 := >= 0.0.3 < 0.0.4
^0.0   := >= 0.0.0 < 0.1.0
^0     := >= 0.0.0 < 1.0.0
```

A version without a sigil or inequality is assumed to be a caret constraint.

10.1.4 Tilde constraints

Tilde constraints are slightly stricted than caret constraints. If a tilde constraint specifies a major and minor version, only changes in the patch version are allowed. If only a major version is specified, changes in the minor and patch versions are allowed.

```
~1.2.3 := >= 1.2.3 < 1.3.0
~1.2   := >= 1.2.0 < 1.3.0
~1     := >= 1.0.0 < 2.0.0
~0.2.3 := >= 0.2.3 < 0.3.0
~0.2   := >= 0.2.0 < 0.3.0
~0.0.3 := >= 0.0.3 < 0.1.0
```

(continues on next page)

(continued from previous page)

```
~0.0    := >= 0.0.0 < 0.1.0
~0      := >= 0.0.0 < 1.0.0
```

10.1.5 The `any` constraint

If a package doesn't care about what version of a package it uses (which it really should; it's impossible to guarantee infinite perpetual forwards compatibility with a package), the `any` constraint can be used, which satisfies every version.

10.1.6 Combining constraints with unions

Multiple constraints can be combined to form a larger constraint by placing a comma in between each constraint, like so: `1.0.0, 2.0.0, >= 3.1.3 <= 3.1.3`. This constraint represents the **union** between its three component constraints, and it requires that the version has either a major version `1` or `2`, or that it's equal to `3.1.3`.

10.2 Dependency Resolution

Dependency resolution for packages is an extremely hard problem (possibly/probably NP-complete). In order to figure out which versions of a package should be used, elba uses the Pubgrub algorithm to do its dependency resolution.

While all of the gory details of how the algorithm works are available both at that design document and the Pub documentation (where Pubgrub was first implemented), the main consequence of this decision is that **only one version of a package can be used at a time.** If separate packages depend on different incompatible versions of the same package, elba will return an error during dependency resolution and will refuse to continue until the conflict is solved.

On the one hand, this aspect of the dependency resolution system has its fair share of drawbacks:

- "Dependency hell" becomes much harder to avoid, since every dependent package is limited to one and only one version
- Getting an ecosystem to upgrade major versions of a package can be much more challenging, as the entire ecosystem is locked to the "stragglers" stuck on previous versions

However, it does have its advantages:

- Because there will be only one version of a package present at all times, any data structures or functions provided by that package can be used freely across between dependencies without fear of incompatibile data structures due to version differences
- Restricting users to one version of a package simplifies module name conflicts

Additionally, one benefit that elba gains from using the Pubgrub algorithm is that elba can provide extremely clear error reporting to help pinpoint and fix the conflict in question. For example, given a dependency tree that looks like this:

- `conflict_simple/root|1.0.0` depends on `conflict_simple/foo ^1.0.0` and `conflict_simple/baz ^1.0.0`.
- `conflict_simple/foo|1.0.0` depends on `conflict_simple/bar ^2.0.0`.
- `conflict_simple/bar|2.0.0` depends on `conflict_simple/baz ^3.0.0`.
- `conflict_simple/baz|1.0.0` and `3.0.0` have no dependencies.
- All these packages are located at the index `index+dir+/index/`.

elba will print the following output when trying to build it:

```
$ elba build
snip...
[error] version solving has failed

Because conflict_simple/bar@index+dir+/index/ any depends on
conflict_simple/baz@index+dir+/index/ >=3.0.0 <4.0.0,
conflict_simple/baz@index+dir+/index/ <!3.0.0, >=!4.0.0 is impossible.
And because conflict_simple/root@index+dir+/index/ >=1.0.0 <=1.0.0 depends
on conflict_simple/baz@index+dir+/index/ >=1.0.0 <2.0.0,
conflict_simple/root@index+dir+/index/ >=1.0.0 <=1.0.0 is impossible.
```

Nice!

CHAPTER 11

The Global Cache

elba uses an internal global cache to store downloaded packages, build packages in a temporary clean directory, and store built packages for future re-use. The structure of the global cache looks like the following:

```
this directory is platform specific:
- Linux: ~/.cache/elba
- Windows: %LOCALAPPDATA%\elba
- macOS: /Users/<user>/Library/Caches/elba
|
|-- build
|   |-- a78bu877c78deadbeef...
|   +-- # snip
|-- indices
|   |-- d3237be53e69715112f...
|   +-- # snip
|-- src
|   |-- d2e4a311d3323b784ef...
|   +-- # snip
+-- tmp
    |-- a78bu877c78deadbeef...
    +-- # snip
```

11.1 Installed binaries

Binaries are special in that they get their own folder separate from the internal cache stuff. Ordinarily this is stored at `~/.elba/bin` for all systems, but this can be controlled in the config, separate from the cache dir. Deleting the whole folder should be safe, but deleting individual binaries might not be; if you try to uninstall them later down the line, you might get an error.

11.2 Folder structure

11.2.1 `build`

This folder stores the binary (i.e. `.ibc` file) outputs of library builds. elba globally caches the builds of all dependencies to avoid having to rebuild the same library over and over across different projects. Each built version of a package gets its own hash which encapsulates the entire environment under which the package was built (package dependencies, etc.), ensuring reproducible builds. This emulates the Nix package manager in some respects.

This folder and its subfolders are safe to delete, although it may cause rebuilds of some packages.

11.2.2 `indices`

This folder stores the downloaded package indices as specified in elba's *configuration*, with a hash corresponding to each different package index.

This folder and its subfolders are safe to delete; elba will redownload any needed indices on its next invocation.

11.2.3 `src`

This folder stores the downloaded sources of packages. elba globally caches these to avoid having to redownload the same files over and over again.

This folder and its subfolders are safe to delete, although it may cause having to redownload and rebuild some packages.

11.2.4 `tmp`

This folder is a temporary build directory for packages, and is more of an implementation detail than anything else. Folders correspond to build hashes for packages, and the internal structure of these folders mirrors the `target/` directory of a local package build.

This folder and its subfolders can be safely deleted.

11.3 Cleaning the cache

. . . can be accomplished with the following invocation:

```
$ elba clean
```

Doing so clears the `artifacts`, `build`, `indices`, `src`, and `tmp` directories.

CHAPTER 12

Indices and tables

- genindex
- modindex
- search