CPM User's Manual

Jonas Oberschweiber Michael Hanus

Institut für Informatik, CAU Kiel, Germany

packages@curry-language.org

April 21, 2017

Abstract

This document describes the Curry package manager (CPM), a tool to distribute and install Curry libraries and manage version dependencies between these libraries.

1 Installing the Curry Package Manager

To install and use CPM, a working installation of either the PAKCS[1] compiler in version 1.14.1 or greater, or the KiCS2[2] compiler in version 0.5.1 or greater is required. Additionally, CPM requires *Git*[3], *curl*[4] and *unzip* to be available on the `PATH` during installation and operation. You also need to ensure that your Haskell installations reads files using UTF-8 encoding by default. Haskell uses the system locale charmap for its default encoding. You can check the current value using `System.IO.localeEncoding` inside a `ghci` session.

To install CPM from the sources, enter the root directory of the CPM source distribution. The main executable `curry` of your Curry system must be in your path (otherwise, you can also specify the root location of your Curry system by modifying the definition of `CURRYROOT` in the `Makefile`). Then type `make` to compile CPM which generates a binary called `cpm` in the `bin` subdirectory. Put this binary somewhere on your path.

Afterwards, run `cpm update` to pull down a copy of the central package index to your system.

2 Package Basics

Essentially, a Curry package is nothing more than a directory structure containing a `package.json` file and a `src` directory at its root. The `package.json` file is a JSON file containing package metadata, the `src` directory contains the Curry modules that make up the package.

We assume familiarity with the JSON file format. A good introduction can be found at http://json.org. The package specification file must contain a top-level JSON object with at least the keys `name`, `author`, `version`, `synopsis` and `dependencies`. More possible fields are described in

[1] https://www.informatik.uni-kiel.de/~pakcs/
[2] https://www-ps.informatik.uni-kiel.de/kics2/
[3] http://www.git-scm.com
[4] https://curl.haxx.se

Section 8. A package's name may contain any ASCII alphanumeric character as well as dashes (-) and underscores (_). It must start with an alphanumeric character. The author field is a free-form field, but the suggested format is either a name (`John Doe`), or a name followed by an email address in angle brackets (`John Doe <john.doe@goldenstate.gov>`). Separate multiple authors with commas.

Versions must be specified in the format laid out in the semantic versioning standard:[5] each version number consists of numeric major, minor and patch versions separated by dot characters as well as an optional pre-release specifier consisting of ASCII alphanumerics and hyphens, e.g. `1.2.3` and `1.2.3-beta5`. Please note that build metadata as specified in the standard is not supported.

The synopsis should be a short summary of what the package does. Use the `description` field for longer form explanations.

Dependencies are specified as a nested JSON object with package names as keys and dependency constraints as values. A dependency constraint restricts the range of versions of the dependency that a package is compatible to. Constraints consist of elementary comparisons that can be combined into conjunctions, which can then be combined into one large disjunction – essentially a disjunctive normal form. The supported comparison operators are $<, \leq, >, \geq, =$ and $\sim>$. The first four are interpreted according to the rules for comparing version numbers laid out in the semantic versioning standard. $\sim>$ is called the *semantic versioning arrow*. It requires that the package version be at least as large as its argument, but still within the same minor version, i.e. $\sim>$ 1.2.3 would match 1.2.3, 1.2.9 and 1.2.55, but not 1.2.2 or 1.3.0.

To combine multiple comparisons into a conjunction, separate them by commas, e.g. $\geq 2.0.0, < 3.0.0$ would match all versions with major version 2. Note that it would not match *2.1.3-beta5* for example, since pre-release versions are only matched if the comparison is explicitly made to a pre-release version, e.g. $=$ 2.1.3-beta5 or $\geq$ 2.1.3-beta2.

Conjunctions can be combined into a disjunction via the || characters, e.g. $\geq 2.0.0, < 3.0.0 || \geq 4.0.0$ would match any version within major version 2 and from major version 4 onwards, but no version within major version 3.

3 Using Packages

Curry packages can be used as dependencies of other Curry packages or to install applications implemented with a package. In the following we describe both possibilities of using packages.

3.1 Creating New Packages

Creating a new Curry package is easy. To use a Curry package in your project, create a `package.json` file in the root, fill it with the minimum amount of information discussed in the previous session, and move your Curry code to a `src` directory inside your project's directory. Alternatively, if you are starting a new project, use the `cpm new <project-name>` command, which creates a new project directory with a `package.json` file for you.[6] Declare a dependency inside the new `package.json` file, e.g.:

[5] http://www.semver.org

[6] The `new` command also creates some other useful template files. Look into the output of this command.

```
{
  ...,
  "dependencies": {
    "json": "~> 1.1.0"
  }
}
```

Then run `cpm install` to install all dependencies of the current package and start your interactive Curry environment with `cpm curry`. You will be able to load the JSON package's modules in your Curry session.

3.2 Installing and Updating Dependencies

To install the current package's dependencies, run `cpm install`. This will install the most recent version of all dependencies that are compatible to the package's dependency constraints. Note that a subsequent run of `cpm install` will always prefer the versions it installed on a previous run, if they are still compatible to the package's dependencies. If you want to explicitly install the newest compatible version regardless of what was installed on previous runs of `cpm install`, you can use the `cpm upgrade` command to upgrade all dependencies to their newest compatible versions, or `cpm upgrade <package>` to update a specific package and all its transitive dependencies to the newest compatible version.

Note that there is also a `cpm update` command, which will update your copy of the central package index to the newest version. You can list all packages of the central package index via the `cpm list` command, or you can search the central package index via the `cpm search` command. See Section 7 for a reference of all commands.

If the package also contains an implementation of a complete executable, e.g., some useful tool, which can be specifed in the `package.json` file (see Section 8), then the command `cpm install` also compiles the application and installs the executable in the `bin` install directory of CPM (see Section 5 for details). The installation of executables can be suppressed by the `cpm install` option `-n` or `--noexec`.

3.3 Checking out Packages

In order to use, experiment with or modify an existing package, one can use the command

```
cpm checkout <package>
```

to install a local copy of a package. This is also useful to install some tool distributed as a package. For instance, to install `curry-genmake`, a tool to generate a `make` file for a Curry application, one can check out the most recent version and install the tool:

```
> cpm checkout makefile
... Package 'makefile-1.3.4' checked out into directory 'makefile'.
> cd makefile
> cpm install
...
INFO  Installing executable 'curry-genmake' into '/home/joe/.cpm/bin'
```

Now, the tool `curry-genmake` is ready to use if `$HOME/.cpm/bin` is in your path (see Section 5 for details about changing the location of this default path).

3.4 Installing Applications of Packages

Some packages do not contain only useful libraries but also application programs or tools. In order to install the executables of such applications without explicitly using the source code of the package, one can use the command

```
cpm installapp <package>
```

This command checks out the package in some internal directory (default: `$HOME/.cpm/app_packages`, see Section 5) and installs the binary of the application provided by the package in `$HOME/.cpm/bin` (see also Section 3.3).

For instance, the most recent version of the web framework Spicey can be installed by the following command:

```
> cpm installapp spicey
... Package 'spicey-xxx' checked out ...
...
INFO  Installing executable 'spiceup' into '/home/joe/.cpm/bin'
```

Now, the binary `spiceup` of Spicey can be used if `$HOME/.cpm/bin` is in your path (see Section 5 for details about changing the location of this default path).

3.5 Executing the Curry Compiler

To use the dependencies of a package, the Curry compiler needs to be started via CPM so that the compiler will know where to search for the modules provided. You can use the `cpm curry` command to start the Curry compiler (which is either the compiler used to install CPM or specified with the configuration option `curry_bin`, see Section 5). Any parameters given to `cpm curry` will be passed along verbatim to the Curry compiler, for example the following will start the Curry compiler, print the result of evaluating the expression `39+3` and then quit.

```
> cpm curry :eval "39+3" :quit
```

To execute other Curry commands such as `curry check` with the package's dependencies available, you can use the `cpm exec` command. `cpm exec` will set the `CURRYPATH` environment variable and then execute the command it is given.

3.6 Replacing Dependencies with Local Versions

During development of your applications, situations may arise in which you want to temporarily replace one of your package's dependencies with a local copy, without having to publish a copy of that dependency somewhere or increasing the dependency's version number. One such situation is a bug in a dependency not controlled by you: if your own package depends on package A and A's current version is 1.0.3 and you encounter a bug in this version, then you might be able to investigate, find and fix the bug. Since you are not the the author of A, however, you cannot release a new version with the bug fixed. So you send off your patch to A's maintainer and wait for 1.0.4

to be released. In the meantime, you want to use your local, fixed copy of version 1.0.3 from your package. The `cpm link` command allows you to replace a dependency with your own local copy.

`cpm link` takes a directory containing a copy of one of the current package's dependencies as its argument. It creates a symbolic link from that directory the the current package's local package cache. If you had a copy of `A-1.0.3` in the `/src/A-1.0.3` directory, you could use `cpm link /src/A-1.0.3` to ensure that any time `A-1.0.3` is used from the current package, your local copy is used instead of the one from the global package cache. To remove any links, use `cpm upgrade` without any arguments, which will clear the local package cache. See Section 6 for more information on the global and local package caches.

4 Authoring Packages

If you want to create packages for other people to use, you should consider filling out more metadata fields than the bare minimum. See Section 8 for a reference of all available fields.

4.1 Semantic Versioning

The versions of published packages should adhere to the semantic versioning standard, which lays out rules for which components of a version number must change if the public API of a package changes. Recall that a semantic versioning version number consists of a major, minor and patch version as well as an optional pre-release specifier. In short, semantic versioning defines the following rules:

- If the type of any public API is changed or removed or the expected behavior of a public API is changed, you must increase the major version number and reset the minor and patch version numbers to 0.
- If a public API is added, you must increase at least the minor version number and reset the patch version number to 0.
- If only bug fixes are introduced, i.e. nothing is added or removed and behavior is only changed to removed deviations from the expected behavior, then it is sufficient to increase the patch version number.
- Once a version is published, it must not be changed.
- For pre-releases, sticking to these rules is encouraged but not required.
- If the major version number is 0, the package is still considered under development and thus unstable. In this case, the rules do not apply, although following them as much as possible as still encouraged. Release 1.0.0 is considered to be the first stable version.

To aid you in following these rules, CPM provides the `diff` command. `diff` can be used to compare the types and behavior of a package's public API between two versions of that package. If it finds any differences, it checks whether they are acceptable under semantic versioning for the difference in version numbers between the two package versions. To use `diff`, you need to be in the directory of one of the versions, i.e., your copy for development, and have the other version installed in CPM's

global package cache (see the `cpm install` command). For example, if you are developing version 1.3.0 of the JSON package and want to make sure you have not introduced any breaking changes when compared to the previous version 1.2.6, you can use the `cpm diff 1.2.6` command while in the directory of version 1.3.0.

CPM will then check the types of all public functions and data types in all exported modules of both versions (see the `exportedModules` field of the package specification) and report any differences and whether they violate semantic versioning. It will also generate a CurryCheck program that will compare the behavior of all exported functions in all exported modules whose types have not changed and execute that program. Note that not all functions can be compared via CurryCheck. In particular, functions taking other functions as arguments (there are a few other minor restrictions) can not be checked so that CPM automatically excludes them from checking.

Note that the results of non-terminating operations, like `Prelude.repeat`, cannot be compared in a finite amount of time. To avoid the execution of possibly non-terminating check programs, CPM compares the behavior of operations only if it can prove the termination or productivity[7] of these operations. Since CPM uses simple criteria to approximate these properties, there might be operations that are terminating or productive but CPM cannot show it. In these cases you can use the compiler pragmas `{-# TERMINATE -#}` or `{-# PRODUCTIVE -#}` to annotate such functions. Then CPM will trust these annotations and treat the annotated operations as terminating or productive, respectively. For instance, CPM will check the following operation although it cannot show its termination:

```
{-# TERMINATE -#}
mcCarthy :: Int  → Int
mcCarthy n | n<=100 = mcCarthy (mcCarthy (n+11))
           | n>100 = n-10
```

As another example, consider the following operation defining an infinite list:

```
ones :: [Int]
ones = 1 : ones
```

Although this operation is not terminating, it is productive since with every step a new constructor is produced. CPM compares such operations by comparing their results up to some depth. On the other hand, the following operation is not classified as productive by CPM (note that it would not be productive if the filter condition is changed to `(>1)`):

```
{-# PRODUCTIVE -#}
anotherOnes :: [Int]
anotherOnes = filter (>0) ones
```

Due to the pragma, CPM will compare this operation as other productive operations.

There might be situations when operations should not be compared, e.g., if the previous version of the operation was buggy. In this case, one can mark those functions with the compiler pragma `{-# NOCOMPARE -#}` so that CPM will not generate tests for them.

[7] An operation is productive if it always produces outermost constructors, i.e., it cannot run forever without producing constructors.

4.2 Publishing a Package

There are three things that need to be done to publish a package: make the package accessible somewhere, add the location to the package specification, and add the package specification to the central package index.

CPM supports ZIP files accessible over HTTP as well as Git repositories as package sources. You are free to choose one of those, but a publicly accessible Git repository is preferred. To add the location to the package specification, use the `source` key. For a HTTP source, use:

```
{
  ...,
  "source": {
    "http": "http://example.com/package-1.0.3.zip"
  }
}
```

For a Git source, you have to specify both the repository as well as the revision that represents the version:

```
{
  ...,
  "source": {
    "git": "git+ssh://git@github.com:john-doe/package.git",
    "tag": "v1.2.3"
  }
}
```

There is also a shorthand, `$version`, available to automatically use a tag consisting of the letter `v` followed by the current version number, as in the example above. Specifying `$version` as the tag and then tagging each version in the format `v1.2.3` is preferred, since it does not require changing the source location in the `package.json` file every time a new version is released.

After you have published the files for your new package version, you have to add the corresponding package specification to the central package index. The central package index is just a Git repository containing a directory for each package, which contain subdirectories for all versions of that package which in turn contain the package specification files. So the specification for version 1.0.5 of the `json` package would be located in `json/1.0.5/package.json`. If you have access to the Git repository containing the central package index, then you can add the package specification yourself. Otherwise, send your package specification file to packages@curry-language.org in order to publish it.

5 Configuration

CPM can be configured via the `$HOME/.cpmrc` configuration file. The following list shows all configuration options and their default values.

`repository_path` The path to the index repository. Default value: `$HOME/.cpm/index`.

`package_install_path` The path to the global package cache. This is where all downloaded packages are stored. Default value: `$HOME/.cpm/packages`

`bin_install_path` The path to the executables of packages. This is the location where the compiled executables of packages containing full applications are stored. Hence, in order to use such applications, one should have this path in the personal load path (environment variable `PATH`). Default value: `$HOME/.cpm/bin`

`app_package_path` The path to the package cache where packages are checked out if only their binaries are installed (see Section 3.4). Default value: `$HOME/.cpm/app_packages`.

`curry_bin` The name of the executable of the Curry system used to compile and test packages. The default value is the binary of the Curry system which has been used to compile CPM.

Note that one can override the values of these configuration options by the CPM options `-d` or `--define`. For instance, to install the binary of the package `spicey` in the directory `$HOME/bin`, one can execute the command

```
> cpm --define bin_install_path=$HOME/bin installapp spicey
```

6 Some CPM Internals

CPM's central package index is a Git repository containing package specification files. A copy of this Git repository is stored on your local system in the `$HOME/.cpm/index` directory, unless you changed the location using the `repository_path` setting. CPM uses the package index when searching for and installing packages and during dependency resolution.

When a package is installed on the system, it is stored in the *global package cache*. By default, the global package cache is located in `$HOME/.cpm/packages`. When a package *foo*, stored in directory `foo`, depends on a package *bar*, a link to *bar's* directory in the global package cache is added to *foo's* local package cache when dependencies are resolved for *foo*. The *local package cache* is stored in `foo/.cpm/package_cache`. Whenever dependencies are resolved, package versions already in the local package cache are preferred over those from the central package index or the global package cache.

When a module inside a package is compiled, packages are first copied from the local package cache to the *run-time cache*, which is stored in `foo/.cpm/packages`. Ultimately, the Curry compiler only sees the package copies in the run-time cache, and never those from the local or global package caches.

7 Command Reference

This section gives a short description of all available CPM commands. In addition to the commands listed here, there is a global parameter `--verbosity` which defaults to `info` but can be increased to `debug` for more output. Furthermore, there is a global parameter `--define` to override the configuration options of CPM, see Section 5.

`info` Gives information on the current package, e.g. the package's name, author, synopsis and its dependency specifications.

`info` *package* `[--all]` Prints information on the newest known version (compatible to the current compiler) of the given package. The option `--all` shows more information.

`info` *package version* `[--all]` Prints basic information on the given package version. The option `--all` shows more information.

`list [--all] [--csv]` List the names and synopses of all packages (compatible to the current compiler) of the central package index. The option `--all` shows also all package versions. The option `--csv` shows the information in CSV format.

`list --category [--csv]` List the category names together with the packages belonging to this category (see Section 8) of the central package index. The option `--csv` shows the information in CSV format.

`search` *query* Searches the names and synopses of all packages (compatible to the current compiler) of the central package index for a term.

`update` Updates the local copy of the central package index to the newest available version.

`install` Installs all dependencies of the current package. Furthermore, if the current package contains an executable application, the application is compiled and the executable is installed (unless the option `-n` or `--noexec` is set).

`install` *package* `[--`*pre*`]` Installs the newest version (compatible to the current compiler) of a package to the global package cache. `--`*pre* enables the installation of pre-release versions.

`install` *package version* Installs a specific version of a package to the global package cache.

`install` *package*`.zip` Installs a package from a ZIP file to the global package cache. The ZIP file must contain at least the package description file `package.json` and the directory `src` containing the Curry source files.

`uninstall` Uninstall the executable installed for this package.

`uninstall` *package version* Uninstalls a specific version of a package from the global package cache.

`checkout` *package* `[--`*pre*`]` Checks out the newest version (compatible to the current compiler) of a package into the local directory *package* in order to test its operations or install a binary of the package. `--`*pre* enables the installation of pre-release versions.

`checkout` *package version* Checks out a specific version of a package into the local directory *package* in order to test its operations or install a binary of the package..

`installapp` *package* `[--`*pre*`]` Install the application provided by the newest version (compatible to the current compiler) of a package. The binary of the application is installed into the directory `$HOME/.cpm/bin` (this location can be changed via the `$HOME/.cpmrc` configuration file or by the CPM option `--define`, see Section 5). `--`*pre* enables the installation of pre-release versions.

`installapp` *package version* Install the application provided by a specific version of a package. The binary of the application is installed into the directory `$HOME/.cpm/bin` (this location can be changed via the `$HOME/.cpmrc` configuration file or by the CPM option `--define`, see Section 5).

`upgrade` Upgrades all dependencies of the current package to the newest compatible version.

`upgrade` *package* Upgrades a specific dependency of the current package and all its transitive dependencies to their newest compatible versions.

`deps` Does a dependency resolution run for the current package and prints out the results. The result is either a list of all package versions chosen or a description of the conflict encountered during dependency resolution.

`test` Tests the current package with CurryCheck. If the package specification contains a definition of a test suite (entry `testsuite`, see Section 8), then the modules defined there are tested. If there is no test suite defined, the list of exported modules are tested, if they are explicitly specified (field `exportedModules` of the package specification), otherwise all modules in the directory `src` (including hierarchical modules stored in its subdirectories) are tested. Using the option `--modules`, one can also specify a comma-separated list of module names to be tested.

`diff` [*version*] Compares the API and behavior of the current package to another version of the same package. If the version option is missing, the latest version of the current package found in the repository is used for comparison. If the options `--api-only` or `--behavior-only` are added, then only the API or the behavior are compared, respectively. Using the option `--modules`, one can also specify a comma-separated list of module names to be compared. Without this option, all exported modules are compared.

As described in Section 4.1, CPM uses property tests to compare the behavior of different package versions. In order to avoid infinite loops durings these tests, CPM analyzes the termination behavior of the involved operations. Using the operation `--unsafe`, CPM omits this program analysis but then you have to ensure that all operations are terminating (or you can annotate them by pragmas, see Section 4.1).

`exec` *command* Executes an arbitrary command with the `CURRYPATH` environment variable set to the paths of all dependencies of the current package. For example, it can be used to execute "`curry check`" or "`curry analyze`" with correct dependencies available.

`curry` *args* Executes the Curry compiler with the dependencies of the current package available. Any arguments are passed verbatim to the compiler.

`link` *source* Can be used to replace a dependency of the current package using a local copy, see Section 3.6 for details.

`clean` Cleans the current package from the generated auxiliariy files, e.g., intermediate Curry files, installed dependent packages, etc. Note that a binary installed in the CPM `bin` directory (by the `install` command) will not be removed. Hence, this command can be used to clean an application package after installing the application.

`new` *project* Creates a new project package with the given name and some template files.

8 Package Specification Reference

This section describes all metadata fields available in a CPM package specification. Mandatory fields are marked with a `*` character.

`name*` The name of the package. Must only contain ASCII letters, digits, hyphens and underscores. Must start with a letter.

`version*` The version of the package. Must follow the format for semantic versioning version numbers.

`author*` The package's author. This is a free-form field, the suggested format is either a name or a name followed by an email address in angle brackets – e.g. `John Doe <john@doe.com>`. Multiple authors should be separated by commas.

`maintainer` The current maintainers of the package, if different from the original authors. This field allows the current maintainers to indicate the best person or persons to contact about the package while attributing the original authors.

`synopsis*` A short form summary of the package's purpose. It should be kept as short as possible (ideally, less than 100 characters).

`description` A longer form description of what the package does.

`category` A list of keywords that characterize the main area where the package can be used, e.g., `Data`, `Numeric`, `GUI`, `Web`, etc.

`license` The license under which the package is distributed. This is a free-form field. In case of a well-known license such as the GNU General Public License[8], the SPDX license identifier[9] should be specified. If a custom license is used, this field should be left blank in favor of the license file field.

`licenseFile` The name of a file in the root directory of the package containing explanations regarding the license of the package or the full text of the license. The suggested name for this file is `LICENSE`.

`copyright` Copyright information regarding the package.

`homepage` The package's web site. This field should contain a valid URL.

`bugReports` A place to report bugs found int he package. The suggested formats are either a valid URL to a bug tracker or an email address.

`repository` The location of a SCM repository containing the package's source code. Should be a valid URL to either a repository (e.g. a Git URL), or a website representing the repository.

[8] https://www.gnu.org/licenses/gpl-3.0.en.html
[9] https://spdx.org/licenses/

`dependencies*` The package's dependencies. This must be JSON object where the keys are package names and the values are version constraints. See Section 2.

`compilerCompatibility` The package's compatibility to different Curry compilers. Expects a JSON object where the keys are compiler names and the values are version constraints. Currently, the supported compiler names are `pakcs` and `kics2`. If this field is missing or contains an empty JSON object, the package is assumed to be compatible to all compilers in all versions.

The compiler compatibility of a package is also relevant when some version of a package should be examined or installed (with CPM commands `info`, `checkout`, `install`, `installapp`). If a newest package should be installed, i.e., no specific version number is provided, then only the newest version which is compatible to the current Curry compiler (see also Section 5 for configuration option `curry_bin`) is considered. Similarly, the current package is executed (CPM commands `curry` and `test`) only if the current Curry compiler is compatible to this package.

`source` A JSON object specifying where the version of the package described in the specification can be obtained. See Section 4.2 for details.

`sourceDirs` A list of directories inside this package where the source code is located. When the package is compiled, these directories are put at the front of the Curry load path. If this field is not specified, `src` is used as the single source directory.

`exportedModules` A list of modules intended for use by consumers of the package. These are the modules compared by the `cpm diff` command (and tested by the `cpm test` command if a list of test modules is not provided). Note that modules not in this list are still accessible to consumers of the package.

`configModule` A module name into which some information about the package configuration (location of the package directory, name of the executable, see below) is written when the package is installed. This could be useful if the package needs some data files stored in this package during run time. For instance, a possible specification could be as follows:

```
{
  ...,
  "configModule": "CPM.PackageConfig",
  ...
}
```

In this case, the package configuration is written into the Curry file `src/CPM/PackageConfig.curry`.

`executable` A JSON object specifying the name of the executable and the main module if this package contains also an executable application. The name of the executable must be defined (with key `name`) whereas the name of the main module (key `main`) is optional. If the latter is missing, CPM assumes that the main module is `Main`. For instance, a possible specification could be as follows:

```
{
```

```
  ...,
  "executable": {
    "name": "cpm",
    "main": "CPM.Main"
  }
}
```

If a package contains an `executable` specification, the command `cpm install` also compiles the main module and installs the executable in the `bin` install directory of CPM (see Section 5 for details).

`testsuite` A JSON object specifying a test suite for this package. This object contains a directory (with key `src-dir`) in which the tests are executed. Furthermore, the test suite must also define a list of modules to be tested (with key `modules`). For instance, a possible test suite specification could be as follows:

```
{
  ...,
  "testsuite": {
    "src-dir": "test",
    "modules": [ "testDataConversion", "testIO" ]
  }
}
```

All these modules are tested with CurryCheck by the command `cpm test`. If no test suite is defined, all (exported) modules are tested in the directory `src`. A test suite can also contain a field `options` which defines a string of options passed to the call to CurryCheck.

If a test suite contains a specific test script instead modules to be tested with CurryCheck, then one can specify the name of this test script in the field `script`. In this case, this script is executed in the test directory (with the possible `options` value added). The script should return the exit code `0` if the test is successful, otherwise a non-zero exit code. Note that one has to specify either a (non-empty) list of modules or a test script name in a test suite, but not both.

One can also specify several test suites for a package. In this case, the `testsuite` value is an array of JSON objects as described above. For instance, a test suite specification for tests in the directories `test` and `examples` could be as follows:

```
{
  ...,
  "testsuite": [
     { "src-dir": "test",
       "options": "-v",
       "script": [ "test.sh" ]
     },
     { "src-dir": "examples",
       "options": "-m80",
       "modules": [ "Nats", "Ints" ]
     }
```

```
    ]
}
```

9 Error Recovery

There might occur situations when your package or repository is in an inconsistent state, e.g., when you manually changed some internal files or such files have been inadvertently changed or deleted, or a package is broken due to an incomplete download. Since CPM checks these files, CPM might exit with an error message that something is wrong. In such cases, it might be a good idea to clean up your package file system. Here are some suggestions how to do this:

`cpm clean`
: This command cleans the current package from generated auxiliariy files (see Section 7). Then you can re-install the package and packages on which it depends by the command `cpm install`.

`rm -rf $HOME/.cpm/packages`
: This cleans all packages which have been previously installed in the global package cache (see Section 6). Such an action might be reasonable in case of some download failure. After clearing the global package cache, all necessary packages are downloaded again when they are needed.

`rm -rf $HOME/.cpm/index`
: This removes the central package index of CPM (see Section 6). You can simply re-install the newest version of this index by the command `cpm update`.