OPAM: a Package Management Systems for OCaml
Version 1.0.0 Roadmap

THIS DOCUMENT IS A DRAFT

Thomas GAZAGNAIRE
thomas.gazagnaire@ocamlpro.com

April 27, 2012

Contents

Overview

This document specifies the design of a package management system for OCaml (OPAM). For the first version of OPAM, we have tried to consider the simplest design choices, even if these choices restrict user possibilities (but we hope not too much). Our goal is to propose a system that we can build in a few months. Some of the design choices might evolve to more complex tasks later, if needed.

A package management system has typically two kinds of users: *end-users* who install and use packages for their own projects; and *packagers*, who create and upload packages. End-users want to install on their machine a consistent collection of *packages* – a package being a collection of OCaml libraries and/or programs. Packagers want to take a collection of their own libraries and programs and make them available to other developpers.

This document describes the fonctional requirements for both kinds of users.

Conventions

In this document, `$home`, `$opam` and `$package` are assumed to be defined as follows:

- `$home` refers to the end-user home path, typically `/home/thomas/` on linux, `/Users/thomas/` on OSX `C:\Documents and Settings\thomas\` on Windows.
- `$opam` refers to the filesystem subtree containing the client state. Default directory is `$home/.opam`.
- `$package` refers to a path in the packager filesystem, where lives the collection of libraries and programs he wants to package.

User variables are written in capital letters, prefixed by `$`. For instance package names will be written `$NAME`, package versions `$VERSION`, and the version of the ocaml compiler currently installed `$OVERSION`.

1 Milestone 1: Foundations

The first milestone of OPAM focuses on providing a limited set of features, dedicated to package management of OCaml packages. OPAM rely on external tools to compile and provide full configuration options to the build tools. The goal for this first milestone is to be as much as possible compatible with any existing build system (including `ocamlfind` and `oasis`) modulo few modifications.

1.1 Client state

The client state is stored on the filesystem, under `$opam`:

1.1.1 Global state

- `$opam/config` is the main configuration file. It defines the OPAM version, the repository addresses and the current compiler version. The file format is described in §1.2.4.
- `$opam/opam/$NAME.$VERSION.opam` is the OPAM specification for the package `$NAME` with version `$VERSION` (which might not be installed). The format of OPAM files is described in §1.2.5.
- `$opam/descr/$NAME.$VERSION` contains the description for the version `$VERSION` of package `$NAME` (which might not be installed). The first line of this file is the package synopsis.
- `$opam/archives/$NAME.$VERSION.tar.gz` contains the source archives for the version `$VERSION` of package `$NAME`.

1.1.2 Compiler-specific state

All the configurations files, libraries and binaries related to the specific `$OVERSION` of the OCaml compiler are stored in `$opam/$OVERSION`.

- `$opam/$OVERSION/installed` is the list of installed packages for the compiler version `$OVERSION`. The file format is described in §1.2.1.
- `$opam/$OVERSION/config/$NAME.config` is a platform-specific configuration file of for the installed package `$NAME` with the compiler version `$OVERSION`. The file format is described in §1.2.6. `$opam/$OVERSION/config/` can be shortened to `$config/` for more readability.
- `$opam/$OVERSION/install/$NAME.install` is a platform-specific package installation file for the installed package `$NAME` with the compiler version `$OVERSION`. The file format is described in §1.2.7. `$opam/$OVERSION/install` can be shortened to `$install/` for more readability.
- `$opam/$OVERSION/lib/$NAME/` contains the libraries associated to the installed package `$NAME` with the compiler version `$OVERSION`. `$opam/$OVERSION/lib/` can be shortened to `$lib/` for more readability.
- `$opam/$OVERSION/doc/$NAME/` contains the documentation associated to the installed package `NAME` with the compiler version `$OVERSION`. `$opam/OVERSION/doc/` can be shortened to `$doc/` for more readability.
- `$opam/$OVERSION/bin/` contains the program files for all installed packages with the compiler version `$OVERSION`. `$opam/$OVERSION/bin/` can be shortened to `$bin/` for more readability.
- `$opam/$OVERSION/build/$NAME.$VERSION/` is a tempory folder used to build package `$NAME` with version `$VERSION`, with compiler version `$OVERSION`. `$opam/$OVERSION/build/` can be shortened to `$build/` for more readability.

- `$opam/$OVERSION/reinstall` contains the list of packages which has been changed upstream since the last upgrade. This can happen for instance when a packager uploads a new archive or fix the OPAM file for a specific package version. Every package appearing in this file will be reinstalled (or upgraded if a new version is available) during the next upgrade when the current version of the compiler is `$OVERSION`. The file format is similar to the one described in §1.2.1.

1.1.3 Repository-specific state

Configuration files for OPAM repositories `REPO` are stored in `$opam/repo/$REPO`. Repositories can be of different kinds (stored on the local filesystem, available via HTTP, available using a custom binary protocol, stored under git, ...); they all share the same base filesystem which is initialized using the `opam-<kind>-init` script (see §1.4.1) and use only the `opam-<kind>-update` (see §1.4.2) and `opam-<kind>-upload` (see §1.4.4) scripts to exchange data between the client and the corresponding OPAM repository.

- `$opam/repo/index` contains the location of packages (ie. in which repositories they can be found and the priority between repositories). The file format is described in 1.2.2.
- `$opam/repo/$REPO/kind` contains the kind associated to the OPAM repository `$REPO` have. The kind is stored as a single word (containing only letters and digits) and specifies which `opam-<kind>-*` scripts to call when updating and uploading this repository.
- `$opam/repo/$REPO/address` contains the address of the OPAM repository `$REPO`. This address is passed as argument to the `opam-<kind>-*` scripts.
- `$opam/repo/$REPO/opam/$NAME.$VERSION.opam` is the OPAM specification for the package `$NAME` with version `$VERSION` (which might not be installed). The format of OPAM files is described in §1.2.5.
- `$opam/repo/$REPO/descr/$NAME.$VERSION` contains the textual description for the version `$VERSION` of package `$NAME` (which might not be installed). The first line of this file is the package synopsis.
- `$opam/repo/$REPO/archives/$NAME.$VERSION.tar.gz` contains the source archives for the version `$VERSION` of package `$NAME`. This folder is populated by calling the corresponding `opam-<kind>-download` script (see §1.4.3).
- `$opam/repo/$REPO/updated` contains the new available packages which have not yet been synchronized with the client state. This file is created by the `opam-<kind>-update` script (see §1.4.2). If the file empty, this means that the client state is up-to-date. The file format is the same as the one described in §1.2.1.
- `$opam/repo/$REPO/upload/$NAME.$VERSION/` contains the OPAM, description and archive files to upload to the OPAM repository for the version `$VERSION` of package `$NAME`. The script `opam-<kind>-update` script (see §1.4.4) read the contents of `upload/` and send it to the repository (if the repository support upload).

1.2 File syntax

1.2.1 List of packages

The following configuration files: `$opam/$OVERSION/installed`, `$opam/$OVERSION/reinstall`, and `$opam/repo/$REPO/updated` follow a very simple syntax. The file is a list of lines which

contains a space-separated name and a version. Each line `$NAME $VERSION` means that the version `$VERSION` of package `$NAME` has been compiled with OCaml version `$OVERSION` and has been installed on the system in `$lib/$NAME` and `$bin/`.

For instance, if `batteries` version `1.0+beta` and `ocamlfind` version `1.2` are installed, then `$opam/$OVERSION/installed` will contain:

```
batteries 1.0+beta
ocamlfind 1.2
```

1.2.2 Index of packages

`$opam/repo/index` follows a very simple syntax: each line of the file contains a space separated list of words `$REPO $NAME $VERSION` specifying that the version `$VERSION` of package `$NAME` is available in the OPAM repository `$REPO`. The file contains information on all available packages (e.g. not only on the installed one).

For instance, if `batteries` version `1.0+beta` is available in the `testing` repository and `ocamlfind` version `1.2` is available in the `default` and testing repositories (where `default` is one being used), then `$opam/repo/index` will contain:

```
testing batteries 1.0+beta
default ocamlfind 1.2
testing ocamlfind 1.2
```

1.2.3 General syntax

Most of the files in the client and server states share the same syntax defined in this section.

Base types The base types for values are:

- `BOOL` is either `true` or `false`
- `STRING` is a doubly-quoted OCaml string, for instance: `"foo"`, `"foo-bar"`, ...
- `SYMBOL` contains only non-letter and non-digit characters, for instance: `=`, `<=`, ... Some symbols have a special meaning and thus are not valid `SYMBOL`s: "`( ) [ ] { } :`".
- `IDENT` starts by a letter and is followed by any number of letters, digit and symbols, for instance: `foo`, `foo-bar`, ...

Compound types Types can be composed together to build more complex values:

- `X Y` is a space-separated pair of value.
- `X | Y` is a value of type either `X` or `Y`.
- `?X` is zero or one occurrence of a value of type `X`.
- `X+` is a space-separated list of values of at least one value of type `X`.
- `X*` is a space-separated list of values of values of type `X` (it might contain no value).

All structured OPAM files share the same syntax:

```
<file>  := <item>*

<item>  := IDENT : <value>
         | ?IDENT: <value>
         | IDENT STRING { <item>+ }

<value> := BOOL
         | STRING
         | SYMBOL
         | IDENT
         | [ <value>+ ]
         | value ( <value>+ )
```

1.2.4 Global configuration file

`$opam/config` follows the syntax defined in §1.2.3 with the following restrictions:

```
<file> :=
    opam-version: "1.0"
    ?repositories: [ <repo>+ ]
    ocaml-version: STRING

<repo> := STRING ( STRING )
```

The field `repositories` contains the list of OPAM repositories with their kind.
The field `ocaml-version` corresponds to the current OCaml compiler (available in the path).

1.2.5 Package specification files: `.opam`

`$opam/opam/$NAME.$VERSION.opam` follows the syntax defined in §1.2.3 with the following restrictions:

```
<file> :=
    opam-version: 1.0
    package STRING {
      version:    STRING
      maintainer: STRING
      ?subst:     [ STRING+ ]
      ?build:     [ command+ ]
      ?depends:   <formula>
      ?conflicts: <formula>
      ?libraries: [ STRING+ ]
      ?syntax:    [ STRING+ ]
    }

<formula>    := STRING
              | STRING ( <constraint> )
              | <formula> '|' <formula>
              | <formula> '&' <formula>
              | ( <formula> )

<constraint> := <comp> STRING
              | <constraint> '|' <constraint>
              | <constraint> '&' <constraint>
```

```
               | ( <constraint> )

<comp>       := '=' | '<' | '>' | '>=' | '<='
<command>    := [ STRING+ ]
```

- The first line specifies the OPAM version.
- The string after `package` should not contain any dot (`'.'`) nor space (`' '`).
- The contents of `version` is `$VERSION`. The content of `maintainer` is the contact address of the package maintainer.
- The content of `subst` is the list of files to substitute variable (see §1.2.8 for the file format and §1.3.7 for the semantic of file substitution).
- The content of `build` is the list of commands to run in order to build the package libraries. The build script should build all the libraries and syntax extensions exported by the package and it should produce the platform-specific configuration and install files (e.g. `$NAME.config` and `$NAME.install`, see §1.2.6 and §1.2.7).
- The `depends` and `conflicts` fields contain formulas over package names, optionally parametrized by version constraints. An expression is either:
  - A package name: `"foo"`;
  - A package name with version constraints: `"foo" (>= "1.2" & <= "3.4")`
  - A disjunction of formulas: `E | F`
  - A conjunction of formulas: `E & F`
  - A formula with parenthesis: `( E )`

  For instance `"foo" (<= "1.2") & ("bar" | "gna" (= "3.14"))` is a valid formula whose semantic is: *any version of package* `"foo"` *lesser or equal to* 1.2 *and either any version of package* `"bar"` *or the version* 3.14 *of package* `"gna"`.
- The `libraries` and `syntax` fields contain the libraries and syntax extensions defined by the package.

1.2.6 Package configuration files: `.config`

`$opam/OVERSION/config/NAME.config` follows the syntax defined in §1.2.3, with the following restrictions:

```
<file>    := <item>*
<item>    := <def> | <section>
<section> :=
    <kind> STRING {
      ?asmcomp:  [ STRING+ ]
      ?bytecomp: [ STRING+ ]
      ?asmlink : [ STRING+ ]
      ?bytelink: [ STRING+ ]
      ?requires: [ <dep>+ ]
      <def>*
```

```
    }
<dep>     := STRING
           | STRING ( STRING+ )
<kind>    := library | syntax
<def>     := IDENT: BOOL
           | IDENT: STRING
           | IDENT: [ STRING+ ]
```

`$NAME.config` contains platform-dependent information which can be useful for other libraries or syntax extensions that want to use libraries defined in the package `$NAME`.

Local and global variables The definitions "`IDENT: BOOL`", "`IDENT: STRING`" and "`IDENT: [ STRING+ ]`", are used to defined variables associated to this package, and are used to substitute variables in template files (see §1.2.8):

- `%{$NAME}$VAR%` will refer to the variable `$VAR` defined at the root of the configuration file `$config/NAME.config`.
- `%{$NAME.$LIB}$VAR%` will refer to the variable `$VAR` defined in the `library` or `syntax` section named `$LIB` in the configuration file `$config/$NAME.config`.

Library and syntax sections Each `library` and `syntax` section defines an OCaml library and the specific compilation flags to enable when using and linking with this library.

The distinction between libraries and syntax extensions is only useful at compile time to know whether the options should be used as compilation or pre-processing arguments (ie. should they go on the compiler command line or should they be passed to the `-pp` option). This is the responsibility of the build tool to do the right thing and the `<kind>` of sections is only used for documentation purposes in OPAM.

The available options are:

- `asmcomp` are compilation options to give to the native compiler (when using the `-c` option)
- `bytecomp` are compilation options to give to the bytecode compiler (when using the `-c` option)
- `asmlink` are linking options to give to the native compiler
- `bytlink` are linking options to give to the bytecode compiler
- `requires` is the list of libraries and syntax extensions the current block is depending on. The full list of compilation and linking options is built by looking at the transitive closure of dependencies.

  The contents of `deps` is either:

  - `"foo"` the block is depending on all the syntax extensions and libraries defined in the package `"foo"`; or
  - `"foo" ("bar" "gna")` the block is depending only on the libraries `"bar"` and `"gna"` defined in the package `"foo"`.

1.2.7 Package installation files: `.install`

`$opam/OVERSION/install/NAME.install` follows the syntax defined in §1.2.3 with the following restrictions:

```
<file> :=
    ?lib:  [ STRING+ ]
    ?bin:  [ <mv>+ ]
    ?doc:  [ STRING+ ]
    ?misc: [ <mv>+ ]

<mv> := STRING
      | STRING ( STRING )
```

- Files listed under `lib` are copied to `$lib/$NAME/`.
- Files listed under `bin` are copied to `$bin/` (they can be renamed using `$SRC ( $DST )`; in this case `$SRC` should be a simple filename, ie. it should not start with a directory name).
- Files listed under `doc` are copied to `$doc/$NAME/`.
- Files listed under `misc` should be processed as follows: for each pair `$SRC ( $DST )`, the tool should ask the user if he wants to install `$SRC` to the absolute path `$DST`.

1.2.8 Substitution files

All of the previous files can be generated using a special mode of `opam` which can perform tests and substitutes variables (see §1.3.7 for the exact command to run). Substitution files contains some templates which will be replaced by some contents. The syntax of templates is the following:

- templates such as `%{$NAME}$VAR%` are replaced by the value of the variable `$VAR` defined at the root of the file `$config/NAME.config`.
- templates such as `%{$NAME.$LIB}$VAR%` are replaced by the value of the variable `$VAR` defined in the `$LIB` section in the file `$config/PACKAGE.config$`

1.3 Client commands

1.3.1 Creating a fresh client state

When an end-user starts OPAM for the first time, he needs to initialize `$opam/` in a consistent state. In order to do so, he should run:

```
$ opam init [-kind $KIND] $REPO $ADDRESS
```

Where:

- `$KIND` is the kind of OPAM repository (default is `http`);
- `$REPO` is the name of the repository (default is `default`); and
- `ADDRESS` is the repository address (default is `http://opam.ocamlpro.com/pub`).

This command will:

1. Create the file `$opam/config` (as specified in §1.2.6)
2. Create an empty `$opam/$OVERSION/installed` file.
3. Initialize `$opam/repo/$REPO` by running `'opam-$KIND-init $ADDRESS'` (see §1.4.2). If the script cannot be found in the path, the command should be canceled and should return a well-defined error.
4. Symlink all OPAM and description files (ie. create a symbolic link from every file in `$opam/repo/$REPO/opam/` to `$opam/opam/` and from every file in `$opam/repo/$REPO/descr/` to `$opam/descr/`).
5. Create `$opam/repo/index` and for each version `$VERSION` of package `$NAME` appearing in the repository, append the line `'$REPO $NAME $VERSION'` to the file.
6. Create the empty directories `$opam/archives`, `$lib/`, `$bin/` and `$doc/`.

1.3.2 Listing packages

When an end-user wants to have information on all available packages, he should run:

```
$ opam list
```

This command will parse `$opam/$OVERSION/installed` to know the installed packages, and `$opam/opam/*.opam` to get all the available packages. It will then build a summary of each packages. The description of each package will be read in `$opam/descr/` if it exists.

For instance, if `batteries` version `1.1.3` is installed, `ounit` version `2.3+dev` is installed and `camomille` is not installed, then running the previous command should display:

```
batteries   1.1.3  Batteries is a standard library replacement
ounit     2.3+dev  Test framework
camomille      --  Unicode support
```

1.3.3 Getting package info

In case the end-user wants a more details view of a specific package, he should run:

```
$ opam info $NAME
```

This command will parse `$opam/$OVERSION/installed` to get the installed version of `$NAME`, will process `$opam/repo/index` to get the repository where the package comes from and will look for `$opam/opam/$NAME.*.opam` to get available versions of `$NAME`. It can then display:

```
package: $NAME
version: $VERSION                # '--' if not installed
versions: $VERSION1, $VERSION2, ...
libraries: $LIB1, $LIB2, ...
syntax: $SYNTAX1, $SYNTAX2, ...
repository: $REPO
description:
  $SYNOPSIS

  $LINE1
  $LINE2
  $LINE3
  ...
```

1.3.4 Installing a package

When an end-user wants to install a new package, he should run:

```
$ opam install $NAME
```

This command will:

1. Compute the transitive closure of dependencies and conflicts of packages using the dependency solver (see §1.3.11). If the dependency solver returns more than one answer, the tool will ask the user to pick one, otherwise it will proceed directly. The dependency solver should also mark the packages to recompile.
2. The dependency solver sorts the collections of packages in topological order. Then, for each of them do:
   (a) Check whether the package is already installed by looking for the line `$NAME $VERSION` in `$opam/$OVERSION/installed`. If not, then:
   (b) Look into the archive cache to see whether it has already been downloaded. The cache location is: `$opam/archives/$NAME.VERSION.tar.gz`
   (c) If not, process `$opam/repo/index/` to get the repository `$REPO` where the archive is available, get the repository kind by looking at `$opam/repo/$REPO/kind` and then ask the repository to download the archive by calling `opam-$KIND-download` (see §1.4.3).
   Once this is done, symlink the archive in `$opam/archives`.
   (d) Decompress the archive into `$build/$NAME.$VERSION/`.
   (e) Substitute the required files.
   (f) Run the list of commands to build the package with `$bin` in the path.
   (g) Process `$build/$NAME.$VERSION/$NAME.install` to install the created files. The file format is described in §1.2.7.
   (h) Install the installation file `$build/$NAME.$VERSION/$NAME.install` in `$install/` and the configuration file `$build/$NAME.$VERSION/$NAME.config` in `$config/`.

1.3.5 Updating index files

When an end-user wants to know what are the latest packages available, he will write:

```
$ opam update
```

This command will follow the following steps:

- For each repositories in `$opam/config`, run the appropriate `opam-$KIND-update` script (see §1.4.2).
- For each repositories in `$opam/config`, process `$opam/repo/$REPO/updated` and update `$opam/repo/index`, `$opam/opam/` and `$opam/desc` accordingly (ie. add the right lines in `$opam/repo/index` and create the missing symlinks). Here, the order in which the repositories are specified is important: the first repository containing a given version for a package will be the one providing it (this can be changed manually by editing `$opam/repo/index` later).

- For each line `$REPO $NAME $VERSION` in `$opam/repo/index`, if the version `$VERSION` of package `$NAME` has been modified upstream (ie. if the line `$NAME $VERSION` appears in `$opam/repo/$REPO/$updated`) and if the package is already installed (ie. it appears in `opam/$OVERSION/installed`), then update `$opam/$OVERSION/reinstall` accordingly (for each compiler version `$OVERSION`).

  Packages in `$opam/$OVERSION/reinstall` will be reinstalled (or upgraded if a new version is available) on the next `opam upgrade` (see §1.3.6), with `$OVERSION` being the current compiler version when the upgrade command is run.

- Delete each `$opam/repo/$REPO/$updated`

1.3.6 Upgrading installed packages

When an end-user wants to upgrade the packages installed on his host, he will write:

```
$ opam upgrade
```

This command will:

- Call the dependency solver (see §1.3.11) to find a consistent state where **most** of the installed packages are upgraded to their latest version. Moreover, packages listed in `$opam/$OVERSION/reinstall` will be reinstalled (or upgraded if a new version is available). It will install each non-installed packages in topological order, similar to what it is done during the install step, See §1.3.4.

- Once this is done the command will delete `$opam/$OVERSION/reinstall`.

1.3.7 Getting package configuration

The first version of OPAM contains the minimal information to be able to use installed libraries. In order to do so, the end-user (or the packager) should run:

```
$ opam config -list-vars
$ opam config -var {$NAME}$VAR
$ opam config -var {$NAME.$LIB}$VAR
$ opam config -subst $FILENAME+
$ opam config [-r] -I        $NAME+
$ opam config [-r] -bytecomp $NAME.$LIB+
$ opam config [-r] -asmcomp  $NAME.$LIB+
$ opam config [-r] -bytelink $NAME.$LIB+
$ opam config [-r] -asmlink  $NAME.$LIB+
```

- `-list-vars` will return the list of all variables defined in installed packages (see §1.2.6)

- `-var $var` will return the value associated to the variable `$var`

- `-subst $FILENAME` replace any occurrence of `%{$NAME}$VAR%` and `%{$NAME.$LIB}$VAR%` as specified in §1.2.8 in `$FILENAME.in` to create `$FILENAME`.

- `-I $NAME` will return the list of paths to include when compiling a project using the package `$NAME` (`-r` gives a result taking into account the transitive closure of dependencies).

- `-bytecomp`, `-asmcomp`, `-bytelink` and `-asmlink` return the associated value for the section `$LIB` in the file `$config/$NAME.config` (`-r` gives a result taking into account the transitive closure of all dependencies).

1.3.8 Uploading packages

When a packager wants to create a package, he should:

1. create `$package/$NAME.$VERSION.opam` containing in the format specified in §1.2.5.
2. create a file describing the package
3. make sure the build scripts:
   - build the libraries and packages advertised in `$package/$NAME.$VERSION.opam`
   - generates a valid `$package/$NAME.install` containing the list of files to install (the file format is described in 1.2.7).
   - generates a valid `$package/$NAME.config` containing the configuration flags for libraries exported by this package (the file format is described in 1.2.6).
4. create an archive `$NAME.$VERSION.tar.gz` with the sources he wants to distribute.
5. run the following command:

   ```
   $ opam upload -opam $OPAM -descr $DESC -archive $ARCHIVE -repo $REPO
   ```

   This command will parse `$OPAM` to get the package name and version and it will move `$OPAM$`, `$DESC` and `$ARCHIVE` in `$opam/repo/$REPO/upload/$NAME.$VERSION` (with the right filenames). It will then call `opam-$KIND-upload` to upload the files upstream (see §1.4.4).

   This command will work only for writable repositories; in case of errors (for instance if the repository is read-only) the script returns a non-zero exit code and stores the error message in `$opam/repo/$REPO/error` (see §1.4.5).

1.3.9 Removing packages

When the user wants to remove a package, he should write:

```
$ opam remove $NAME
```

This command will check whether the package `$NAME` is installed, and if yes, it will display to the user the list packages that will be uninstalled (ie. the transitive closure of all forward-dependencies). If the user accepts the list, all the packages should be uninstalled, and the client state should be let in a consistent state.

1.3.10 Managing OPAM repository

When the user wants to manage OPAM repositories, he should write:

```
$ opam repository -list
$ opam repository -rm $REPO
$ opam repository -add [-kind $KIND] $REPO $ADRESS
```

- `-list` lists the current repositories by looking at `$opam/config`

- `-rm $REPO` deletes `$opam/repo/$REPO` and removes `$REPO` from the `repositories` list in `$opam/config`. Then, for each package in `$opam/repo/index` it updates the link between packages and repositories (ie. it either deletes packages or symlink them to the new repository containing the package).
- `-add [-kind $KIND] $REPO $ADDRESS` initializes `$REPO` as described in §1.3.1.

1.3.11 Dependency solver

Dependency solving is a hard problem and we do not plan to start from scratch implementing a new SAT solver. Thus our plan to integrate (as a library) the Debian depency solver for CUDF files, which is written in OCaml.

- the dependency solver should run on the client; and
- the dependency solver should take as input a list of packages (with some optional version information) the user wants to install, upgrade and remove and it should return a consistent list of packages (with version numbers) to install, upgrade, recompile and remove.

1.4 OPAM repository scripts

As stated above, OPAM repositories can be of different kinds. A given kind `$KIND` has associated scripts `opam-$KIND-init`, `opam-$KIND-update`, `opam-$KIND-download` and `opam-$KIND-upload` which should behave in the way specified below.

1.4.1 Init script

```
$ opam-$KIND-init $opam/repo/$REPO $ADDRESS
```

This script should contact the OPAM repository located at address `$ADDRESS` and create `$opam/repo/$REPO` as specified in §1.1.3.

1.4.2 Update script

```
$ opam-$KIND-update $opam/repo/$REPO
```

This script should get the list of newly available packages, by contacting the OPAM repository whose address is specified in `$opam/repo/$REPO/address`. It should then create (or update) `$opam/repo/$REPO/updated` accordingly. It should also create (or update) the OPAM files stored in `$opam/repo/$REPO/opam` and the description files stored in `$opam/repo/$REPO/descr`.

1.4.3 Download script

```
$ opam-$KIND-download $opam/repo/$REPO $NAME $VERSION
```

This script should get `$NAME.$VERSION.tar.gz` from the OPAM repository whose address is stored in `$opam/repo/$REPO/address`. It should then move the archive file into `$opam/repo/$REPO/archives/` and overwrite the previous file if one is already there.

1.4.4 Upload script

```
$ opam-$KIND-init $opam/repo/$REPO
```

This script should send the contents of `$opam/repo/$REPO/upload/` to the repository whose address is stored in `$opam/repo/$REPO/address` and remove the folder contents when it's done.

1.4.5 Error handling

In case any of the above scripts return with a non-zero exit code, the full error message will be stored into `opam/repo/$REPO/error`

1.4.6 RSYNC repository

The first Milestone includes the necessary scripts to support `rsync` repositories. In order to do so:

- The OPAM repository is a filesystem similar to what is described in `$opam/repo/$REPO`.
- `opam-rsync-init $PATH $ADDRESS` simply runs something like:

```
$ cd $PATH && rsync [..] $ADDRESS/opam && rsync [..] $ADDRESS/descr
```

- `opam-rsync-download $PATH $NAME $VERSION` will simply run something like:

```
$ cd $PATH && rsync [..] $ADDRESS/archives/$NAME.$VERSION.tar.gz
```

- `opam-rsync-update $PATH` will also run `rsync` and filter the command output to populate `$opam/repo/$REPO/updated` accordingly.
- `opam-rsync-upload $PATH` will return an error if the OPAM repository is not writable (for instance if the repository if an HTTP server). In this case the server state can be updated manually by copying the new files in the right place on the server filesystem.

2 Milestone 2: OPAM server

The only OPAM repository kind available in Milestone 1 is `http`. This second milestone describes the implementation of an OPAM repository using a custom binary protocol (ie. not relying on HTTP) between a client and a server. This is intended to be the preferred way to upload new packages to OPAM repositories.

2.0.7 Server state

In this section, `$opamserver` refers to the filesystem subtree containing the server state. Default directory is `$home/.opam-server`.

2.0.8 Getting the list of packages

```
val getList   : repo -> (name * version) list
val updateList: repo -> (name * version) list
```

`getList $HOSTNAME` returns the full list of available packages. This command is intended to be run only once, when the repository `$HOSTNAME` is initialized.

`updateList $HOSTNAME` updates the given repository and returns the list of newly available packages. For repositories not using the custom OPAM protocol (eg. not starting by `opam://`) this means running the script `$opamserver/$HOSTNAME/update` which should return a list of lines of (package name, version) pairs using the same format as described in §1.2.1.

2.0.9 Getting OPAM files

```
val getOPAM: repo -> (name * version) -> opam
```

`getOPAM repo (name,version)` returns the corresponding OPAM filename as an absolute location in the filesystem (which should be .

2.0.10 Getting description files

```
val getDescr: repo -> (name * version) -> descr
```

`getDescr repo (name,version)` returns the corresponding description file.

2.0.11 Getting package archive

```
val getArchive: repo -> (name * version) -> archive
```

`getArchive repo (name,version)` returns the corresponding package archive.

2.0.12 Uploading new archives

```
val newArchive: repo -> (opam * archive) -> unit
```

`newArchive(opam,archive)` takes as input an OPAM file and the corresponding package archive, and upload the server state. This function works only for READ-WRITE repository. In case of a READ-ONLY one, a suitable error message is returned to the user.

2.0.13 Binary Protocol

In case of READ-WRITE repositories, the server state can be queried and modified by any OPAM clients, using the following binary protocol

- Communication between clients and servers always start by an hand-shake to agree on the protocol version.
- All the basic values (names, versions and binary data) are represented as OCaml strings.
- More complex values are marshaled using a simple binary protocol: the first byte represents the message number, and then each message argument is stacked in the message with its size as prefix. The list of messages *from the client to server* is:

Client-to-Server Message	Arguments	Description
`GetList`	–	Ask for the list of all OPAM files
`GetOPAM`	`name : string` `version: string`	Ask for the binary representation of a given OPAM file
`GetArchive`	`name : string` `version: string`	Ask for the binary representation of a given archive file

`NewArchive`	`name : string`
	`version: string`
	`opam : string`
	`archive: string`
		Create a new package on the server. The client should provide the OPAM file and the source archive.

`UpdateArchive`	`name : string`
	`version: string`
	`opam : string`
	`archive: string`
	`key : string`
		Update a new version of a given package on the server. The client should also provide a security key

- Answers from the server are encoded in the same way (ie, a byte for the message number, followed by optional arguments prefixed by their size). List arguments are encoded by stacking first the lenght, and then all the elements of the list in sequential order. The list of messages *from servers to clients* is:

Server-to-Client Message	Arguments	Description
`GetList`	`list : (string*string) list`	Return the list of available package names and versions
`GetOPAM`	`opam : string`	Return an OPAM file
`GetArchivwe`	`archive: string`	Return an archive file
`NewArchive`	`key : string`	Return a security key
`UpdateArchive`	–	The update went OK
`Error`	`error : string`	An error occurred

Note that when an error is raised by an arbitrary function at server side, the client receives `Error _`.

3 Milestone 3: Link Information

This milestone focuses on adding the right level of linking information, in order to be able to use packages more easily.

4 Milestone 4: Server Authentication

This version focuses on server authentication.

4.1 RPC protocol

The protocol should be specified (using either a binary format or a JSON format).

4.2 Server authentication

The server should be able to ask for basic credential proofs. The protocol can be sketched as follows:

- packagers store keys in `$opam/keys/NAME`. These keys are random strings of size 128.
- the server stores key hashes in `$opamserver/hashes/NAME`.
- when a packager wants to upload a fresh package, he still uses `newArchive`. However, the return type of this function is changed in order to return a random key. OPAM clients then stores that key in `$opam/keys/NAME`.
- when a packager wants to uplaod a new version of an existing package, he uses the function `val updateArchive: (opam * string * string) -> bool`. `updateArchive` takes as argument an OCaml value representing the OPAM file contents, the archive file as a binary string and the key as a string. The server then checks whether the hash of the key is equal to the one stored in `$opamserver/hashes/NAME`; if yes, it updates the package and return `true`, if no if it returns `false`.
- packager email should be specified in `NAME.opam`:

5 Milestones 6: Pre-Processors Information

This milestone focus on the support of pre-processors.

5.1 Getting package preprocessor options

The user should be able to run:

```
$ opam-config -bytepp NAME
$ opam-config -asmpp NAME
```

This command will return the command line option to build the preprocessor exported by package `NAME`.

In order to do so, packagers should describe exported preprocessors in the corresponding `NAME`.descr:

```
syntax foo {
  requires: bar, gni          // list of syntax dependencies
  pp: -parser o -printer p    // common options to asmpp and bytepp
  bytepp: ...
}
```

6 Milestones 7: Support of Multiple Compiler Versions

This milestone focus on the support of multiple compiler versions.

6.1 Compiler Description Files

For each compiler version `OVERSION`, the client and server states will be extended with the following files:

- `$opam/compilers/OVERSION.comp`
- `$opamserver/compilers/OVERSION.comp`

Each `.comp` file contains:

- the location where this version can be downloaded. It can be an archive available via `http` or using CVS such as `svn` or `git`.
- eventual options to pass to the configure script. `-prefix=$opam/OVERSION/` will be automatically added to these options.
- options to pass to `make`.
- eventual patch address, available via `http` or locally on the filesystem

For instance, `3.12.1+memprof.comp` (OCaml version 3.12.1 with the memory profiling patch) looks like:

```
src:       http://caml.inria.fr/pub/distrib/ocaml-3.12/ocaml-3.12.1.tar.gz
build:     world world.opt
patches:   http://bozman.cagdas.free.fr/documents/ocamlmemprof-3.12.0.patch
```

And `trunk-tk-byte.comp` (OCaml from SVN trunk, with no *tk* support and only in bytecode) looks like:

```
src:       http://caml.inria.fr/pub/distrib/ocaml-3.12/ocaml-3.12.1.tar.gz
configure: -no-tk
build:     world
```

6.2 Milestone 8: Version Pinning

6.3 Milestones 9: Parallel Build

6.4 Milestone 10: Version Comparison Scheme

6.5 Milestone 11: Database of Installed Files