Overview of the CodeGradX Project
Version: 1751

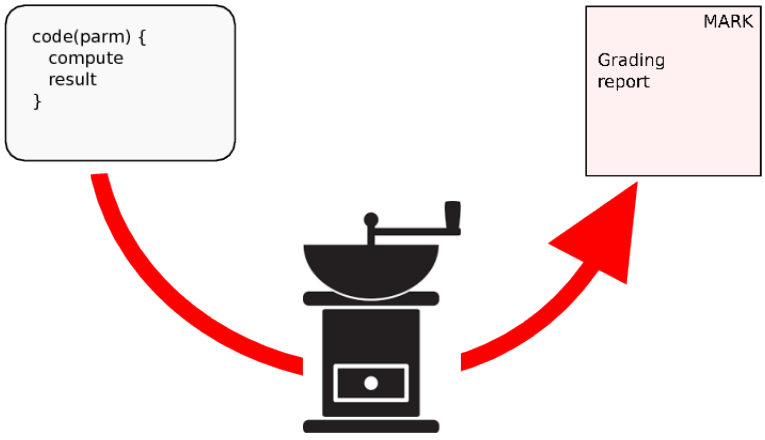
Christian Queinnec
christian.queinnec@codegradx.org

August 21, 2018

This document describes the goals and main features of the CodeGradX project. This project was previously named FW4EX for *Framework for Exercises* but this name cannot be pronounced by a regular human being. However many entities are still named after FW4EX.

This project proposes a platform where teachers may offer exercises (homeworks) to students who may submit (upload) their works. These works are mechanically graded and a report is made available to the student, to their teachers and to the author of the exercise. Thus an exercise not only contains some questions, it contains as well a grading program to evaluate and grade students' works.

This document describes the CodeGradX project, its main features and protocols. Parts of this document describe the rationale under some decisions, it also contains notes for myself.



Contents

Chapter 1

Overview

This chapter exposes some terminology then three different points of view on the CodeGradX project. The points of views are successively taken from a student, a teacher, an author or the CodeGradX project itself. The various servers that are involved in these points of view are described in terms of use cases.

1.1 Terminology

A *student* is someone that has to study an *exercise*. A *teacher* is someone that prescribes an exercise to students. An *author* is someone that designed an exercise. A *job* is one answer written by a student sent to an exercise in order to be graded. Grading a job produces a *report*. A *pseudo-job* is one answer written by the *author* of an exercise whose grade is specified by the *author*. A pseudo-job participates to the auto-test of an exercise. A *batch* is a set of jobs to be graded in one go and returned to a *teacher*.

A *campaign* is a selection of exercises gathered by a *teacher* and given as assignments to a group of students in order to fulfill some goals for a given duration. Some statistics are performed on the basis of campaigns.

1.2 From the student point of view

A student browses a CodeGradX web site and selects an exercise (this exercise may as well be suggested by a teacher), studies the questions proposed in the exercise, writes one or several files (using a web editor, a regular text editor or a more complex IDE) and submits the resulting file(s) to some web site. The CodeGradX infrastructure behind the web site analyzes the files, marks them, annotates them and generates a report (a file) summarizing this analysis. These successive phases are detailed hereafter.

1.2.1 Selection of an exercise

The selection of an exercise is done via a regular browser. The student browses some web site built by some teacher and looks for an interesting exercise. The web site may give advices concerning the selection of a new exercise given the known history of the student, the web site may also hold sets i.e., predefined sets of exercises recommended by teachers.

1.2.2 Downloading accompanying material

Once selected, the student has to be registered and authenticated in order to get the accompanying (program or data) files required to study the exercise. These files may be downloaded via a regular browser. If there are numerous files to download, they are gathered in a single (possibly compressed) archive (a zip file since zip libraries are widely available).

An exercise is an archive containing an XML descriptor that describes it. Texts (and mainly the stem of the exercise) are HTML fragments. See 11 for more details.

1.2.3 Submission

Once the homework is completed, the student has to submit the resulting files to an acquisition server (quite often this is `a.codegradx.org`). These files are gathered in a job and queued towards the grading process. In order to submit jobs towards an exercise, you need to know the URL representing the exercise.

A REST-ish protocol has been designed for this submission, see Section 3 for more details. This protocol may be put to use with a simple web form in a regular browser. Other special clients (for instance, an Eclipse or Emacs plugin) may also be used.

The submission is directed towards a specific URL contained in the descriptor of the exercise. This URL addresses an acquisition server. The main goal of an acquisition server is to be as robust as possible in order to never loose any job corresponding to a registered student and to a proposed exercise. All non conformant submissions should be promptly rejected. Once a job is archived, it is queued towards some grading daemon.

1.2.4 Grading

The grading process analyzes the job. It extracts the grading code from the description of the exercise and runs it against the student's files. There is a specific set of rules for that task, see Appendix 10.

This is a difficult process since the (potentially nefarious) code of the teacher has to analyze the job (containing potentially malicious code) of the student. Eventually, the output of the grader, the *report*, is persistently stored on a storage web server (quite often the `s.codegradx.org`).

1.2.5 Result

Finally, for any job, the student may access the associated report through the web site or his history. Various means allow the student to check whether the jobs are graded and how to obtain the associated reports (mail, web, feed, etc.).

The author of an exercise can also obtain the associated report. Teachers that prescribe this exercise may also obtain the associated reports.

The storage server is very dumb and only serves reports. In particular, it does not depend on the grading process and should be immune to any failure of the other servers. The storage server should be difficult to download recursively.

These reports are anonymous: they do not contain information related to the student's identity. The reports are accessible via the prominent web site or through other means (mail, feed, etc.).

The storage server may be a dedicated server or an Amazon S3 bucket or an OVH cloud.

1.2.6 Conclusion

Figure 1.1 shows the involved servers and their relationship. The most prominent server is the `www.codegradx.org` Web server. With this server, students register themselves, manage their account, choose exercises, supervise their jobs and obtain their report. More services will be added.

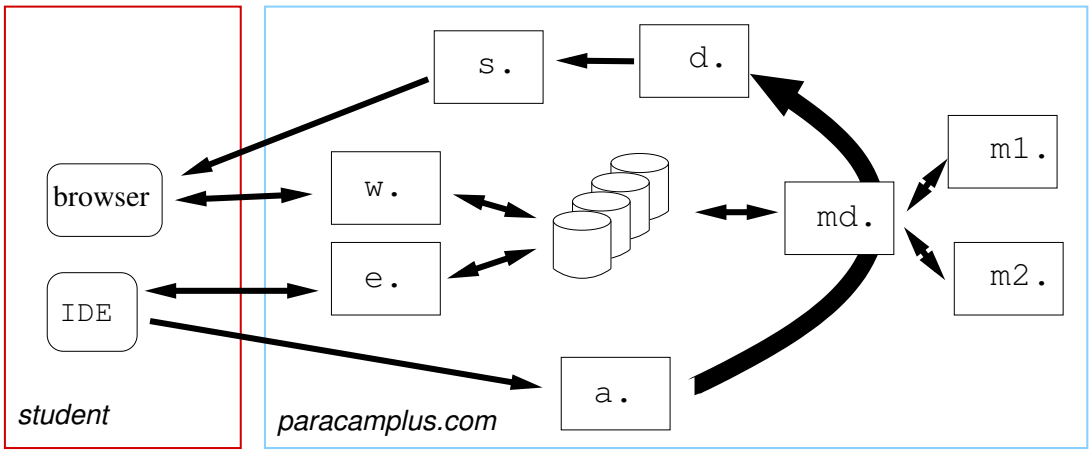


Figure 1.1: Students, servers and their relationships

Besides, the other technical independent servers are:

- the acquisition server accepts jobs,
- the grading daemon handles jobs. This daemon is itself organized as a grading driver controlling grading slaves.
- the exercise server gives information about exercises.
- the storage server stores reports.

The grading daemon is an internal daemon not to be requested directly by students or teachers. The acquisition server is reachable from the net; it should be as robust as possible therefore it is not tied to the database. The storage server is dumb and is not tied to the database. The storage server is monitored by the dynamic storage server.

The exercise server delivers information about exercises. It is not intended for humans. This server is tied to the (rather static) database of exercises.

The dynamic part of the database is only accessible from the complex servers: the web server, the exercise server and the grading daemon(s). This (rather dynamic) database contains the instantaneous state of the users, of the exercises, of the jobs, of the servers, etc.

1.3 From a teacher point of view

A teacher defines a campaign that is, a set of exercises, a set of teachers, a set of students (this set may be open to new students), a starting date and and end date. The teacher can modify the list of exercises. The teacher can also follow the progress of the students.

1.4 From an author point of view

Authors submit an exercise that is, a compressed archive file containing some texts, some questions and the programs to grade jobs from students.

The business model is as follows (except that it is not yet implemented): the author sets the price of the exercise. The exercise will be offered on the web site hosted by the `e.codegradx.org` server as long as its price exceeds the cost of grading a student's job. The author will get the price he sets less the grading cost.

The exercise is uploaded as a compressed (gzip) archive (tar) file. This archive contains a descriptor: a file named `fw4ex.xml`, an XML file satisfying the `fw4ex.rnc` grammar (more precisely the `exercise` description).

An exercise is identified and versioned by its author. It also receives an internal identifier within CodeGradX. The archive file contains the texts of the question(s) (these texts are expressed in a subset of XHTML) and the files (scripts or data) needed to grade the students' submissions.

In order to test that the grading process is correct, an exercise also contains pseudo-jobs whose mark is already known and specified by the author of the exercise. When an exercise is uploaded, the CodeGradX machinery checks that the pseudo-jobs get the expected mark. Two pseudo-jobs are easy to incorporate within an exercise: the *empty* pseudo-jobs does not contain anything, its expected mark should be zero. The *perfect* pseudo-job should get the greatest possible mark. It is recommended to add more pseudo-jobs.

The pseudo-jobs allow the author to be sure that the uploaded exercise is graded according to his wishes. It also serves to check the non-regression of the CodeGradX machinery when this machinery evolves. Finally it also gives an indication of the grading cost so the price may be adjusted: too expensive, the exercise might not be choosed by students, too cheap, the exercise cannot be offered since the grading cost will be a pure loss.

Examples of exercises may be found on the authors' part of the web site.

1.5 From CodeGradX infrastructure point of view

See Figure 1.1 and Figure 1.2 to accompany the explanations below. There are a number of independent but specialized servers.

From 2014, most servers as well as marking drivers and marking slaves are now implemented as Docker containers. All these containers have names prefixed with `paracamplus/aestxyz_`.

1.5.1 Server `p` (for portal)

This is a server presenting the various servers dedicated to a course or to a programming language. The portal manages students' registration allowing them to access the servers of the whole constellation.

1.5.2 Server `www` (for web site)

This is the main server from which students, teachers and authors manage their account and monitor the services they had required. This server contains an evolving and expanding web application.

The `www` server is intended to be the front end Web server. Currently, there are specialized web servers (`js`, `unx`, `scm` for instance) that offer a restricted set of exercises for a restricted set of students).

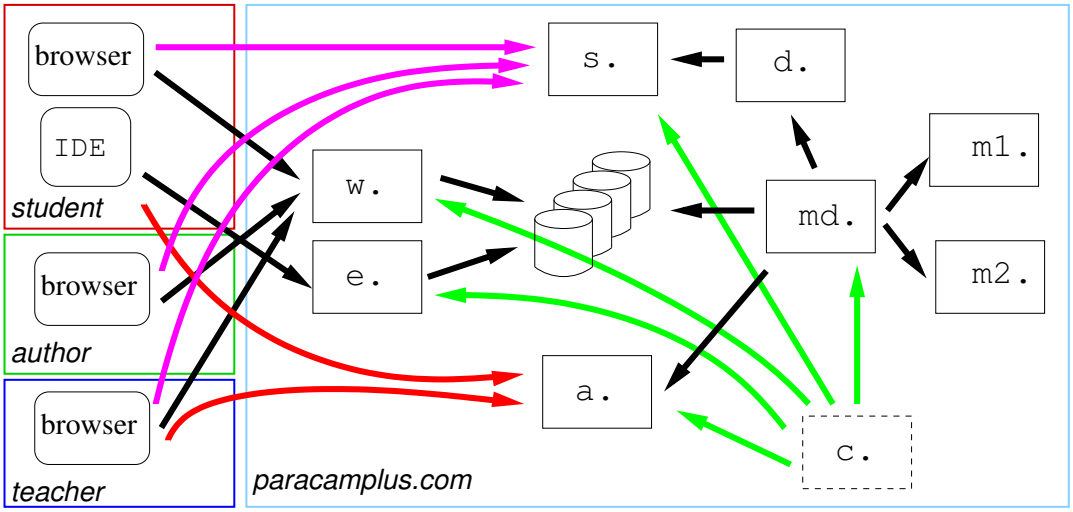


Figure 1.2: CodeGradX infrastructure. The arrow tells which machine initiates connections.

Starting in 2015, these servers do not have access to the central database, they use a JavaScript API.

1.5.3 Server `a` (for acquisition)

This is the server handling submissions from students, turning them into jobs. There is a special protocol to submit files involving a safe cookie and a ciphered identifier for exercises. See chapter 3 for further details.

To cope with failures, there are more than one acquisition server. To date, these are `a0.codegradx.org`, `a1.codegradx.org`, etc. Currently, `a.codegradx.org` is a proxy hiding some acquisition servers.

1.5.4 Daemon `md` (for marking driver)

This daemon regularly polls the acquisition servers and pulls jobs from them. The grading driver instantiates grading slaves, feeds them with jobs, polls them to retrieve the associated reports and eventually transfer jobs and reports to the storage servers.

The grading driver logs facts into the database for the benefit of other servers.

To cope with failures, there are more than one grading server. Jobs are graded concurrently but eventually stored into the same storage server.

1.5.5 Machine `ms` (for marking slave)

These (often virtual) machines grade jobs in the most secure way. They are slaves monitored by the grading driver. They are only accessed via `ssh`.

1.5.6 Server `d` (for dynamic storage)

This server monitors the archival of jobs and reports. It is fed by the multiple concurrent grading drivers and detects marking anomalies if any. It then stores jobs and reports in the various storage servers available. This is an internal server of the CodeGradX constellation.

1.5.7 Server s (for storage)

This server archives jobs and reports. It is fed by the dynamic storage server and accessed by students, authors or teachers. It may be implemented by a regular httpd daemon (Apache, Nginx, etc.) correctly configured or a storage server such as S3 from Amazon or the equivalent from OVH. URLs are UUID-based to avoid directories congestion.

Currently, there are multiple storage servers known with different names such as `s0.codegradx.org`, `s1.codegradx.org`, etc.

1.5.8 Server e (for exercise)

This server archives exercises, it serves them to students. It uses safe identifiers to name exercises within URLs. It also serves sets of exercises elaborated by teachers.

To cope with failures, there are more than one exercise server. To date, these are `e0.codegradx.org` and `e1.codegradx.org`, etc. Server `e.codegradx.org` is a proxy hiding some exercise servers.

1.5.9 Server x (for XML)

This server proposes a number of REST-based services.

Currently, there are more than one such server, they have all acces to a common, shared, single database `x0.codegradx.org`, `x1.codegradx.org` etc. And as usual, `x.codegradx.org` is a proxy hiding some servers.

Chapter 2

Exercise protocol (the E protocol)

This chapter describes how to discover information concerning an exercise. A special server nicknamed `e.codegradx.org` delivers this information since it is rather static and intended for computers rather than for humans. CodeGradX clients (embedded in some IDE such as Eclipse or Emacs) analyze this information and build pages, forms, dialogs to let the student practice the exercise with great comfort.

2.1 Summary

Concerning an exercise, the delivered information contains whatever information required by the student to practice this exercise, to get the accompanying files and to submit one or many answers. This information is delivered as a compressed (zipped) archive containing an XML descriptor (a file named `fw4ex.xml`, an instance of the `exerciseContent` element, see chapter 11) describing the exercise, the accompanying files, the questions and the scripts required to initialize the student's machine (for instance, compile required libraries, uncompress data files, etc.) to let him able to practice the exercise.

One may only perform the following public actions (E identifies an exercise, cf. 3.2). The `e.codegradx.org` server cannot be implemented by a dumb storage server since the E parameter must be deciphered, the user's cookie may also be deciphered).

`GET /alive`	return some JSON data telling if the server is alive
`GET /exerciseload`	return some JSON data telling how many exercises are waiting to be checked

`GET /exercise/E`
`PUT /exercise/E`
`DELETE /exercise/E`
`GET /exercisecontent/E`
	get the whole exercise.tgz
	set the whole exercise.tgz
	delete the whole exercise.tgz
	get the details of an exercise

`GET /exercisecontent/E/content`	get the XML description of the exercise
`GET /exercisecontent/E/stem`	get the XML stem of the exercise
`GET /exercisecontent/E/path/P`	get the equipment file P
`GET /path/S`	get the S set of exercises
`POST /exercises/`	post a new exercise
`GET /summary/UUID`	get the public (if any) summary of the exercise

/alive returns a JSON document with date and version of the server. This allows to check whether the E server is alive.

/exerciseload returns a JSON document with date and load that is, the number of exercises that are waiting to be checked.

/exercise/E for admins only, gets, sets or deletes the whole exercise.tgz.

/exercisecontent/E returns a zip file containing a detailed description of an exercise. This zip file is intended for students to allow them to get the stem and relevant associated files. The MIME type is `application/octet-stream`.

/exercisecontent/E/content returns an XML file (an instance of exerciseContent) describing the exercise. This file corresponds to the fw4ex.xml stored in the previous zip file. This file also contains the stem and the summary as obtained by the next URLs as well as the list of equipment files that may be individually retrieved by an URL below.

/exercisecontent/E/stem returns an XML file containing the stem of the exercise that is, a readable text with an introduction followed by questions. This is a convenient service for the OneLiner framework that displays a question and an input box, submits it to the CodeGradX system, waits for the grading reports and displays it. The stem is an XML document (an instance of exerciseStem) also containing the list of equipment files.

/exercisecontent/E/path/P returns the single P equipment file from the exercise. Equipment files are files given to the student in order to perform the exercise. Of course, the requester should be allowed to retrieve this file.

/path/S returns an XML document (an instance of exercisesPath) describing a set of exercises to perform. Sets are named (often with the name of the corresponding lectures). It may also return an equivalent JSON document.

/exercises/ is used to post a new exercise. Only users blessed as authors may post exercises. This service returns an XML document containing the UUID christening the fresh exercise and the location where the associated report will pop up.

When returning JSON data, JSONP (or JSOD *Javascript on demand*) mode is also supported. The appropriate query string parameter may be named `cb`.

2.1.1 Use of equipment files

Equipment files are given to the students, they accompany the stem. They are often used to provide students with data, libraries, code fragments, etc. They may also be used to provide images that can appear in stems. Each equipment file can be fetched by an URL such as `/exercisecontent/E/path/P`. So a stem containing an `<img src='path/P' />` tag will include the image file `P` within the stem.

2.2 Format

This section details the content of the zip file describing an exercise. This zip file is an extract of the whole tar gzipped file restricted to the sole information needed by students. The zip file contains at its root a descriptor (a file named `fw4ex.xml`: an extract of the original `fw4ex.xml` from the exercise) describing the content of the zip file. Other files may be present, they are referenced from the descriptor. All this information is derived from the original exercise as designed by its author(s). All this information may be disclosed to the student.

Here follows an example of the content of such a zipped file for an imaginary exercise:

```
fw4ex.xml
xml/question1.xml
document.pdf
otherdata/u1.data.gz
otherdata/u2.data.gz
otherdata/u3.data.gz
```

The descriptor contains an `identification` element. This element contains the name of the exercise (as given by its author(s)), its birth date, a summary (an XHTML-like simple text) and the identity of its author(s) along with the email to use if the student may contact the author (by default, this is a CodeGradX internal email address).

The descriptor contains a `conditions` element describing what is required to practice this exercise. The conditions may be financial, may require some specific hardware or software or may even require some special knowledge or abilities.

The descriptor contains a `equipment` element describing all the files of the archive, their digest (to check file integrity), their mode (binary or text (including their encoding)).

The descriptor contains a `initializing` element describing how to install the accompanying files on the student's machine. The installation is a series of scripts to run and files to copy somewhere. Normally files are only installed somewhere under the personal (HOME) directory of the student.

The descriptor contains a `content` element describing the stem of the exercise. The exercise is made of an introduction, a series of questions and followed by a conclusion. Each question has a name, a maximal mark, some expectations (the name of the files that the student should submit) and of course the stem of the question.

2.3 Administrator supplementary interface

Other requests are possible but mostly restricted to administrators.

`PUT /exercisecontent/E`	replaces the student's view of an exercise.
`DELETE /exercisecontent/E`	removes the student's view of an exercise.
`GET /exercise/E`	serves the original tar gzipped file defining an exercise.
`PUT /exercise/E`	replaces the original tar gzipped file defining an exercise with the tar gzipped file contained in the body of the request.
`DELETE /exercise/E`	removes an exercise.
`GET /exercises`	returns the state of the server.
`GET /exercises/`	list all new exercises
`GET /exercises/UUID`	get the exercise submission named UUID
`DELETE /exercises/UUID`	remove the exercise submission named UUID

/exercisecontent/E returns a zip file containing a detailed description of an exercise. The MIME type is `application/octet-stream`.

Using a PUT request towards this URL, allows to replace an exercise (in fact, this is not advised since a new version of an exercise should have a new identifier. This eases checking non regression). However this service may be used in conjunction with the following one to manage the memory space the E server uses. The body of the request should be a zipped file.

/exercise/E returns the original tar gzipped file defining an exercise. Since this is an asset, only administrators may perpetrate this! PUT and DELETE are also offered to manage the memory space used by the E server.

/exercises allows to know the state of the E server. It also allows to list the freshly arrived exercises that must be checked before being offered by the CodeGradX system.

/exercises/UUID allows to get or remove a freshly arrived exercise.

A fresh exercise (a tar gzipped file defining an exercise) is wrapped within an exerciseSubmission (a tar gzipped file containing the original exercise as well as an `fw4ex.xml` containing administrative information). A fresh exercise will be fetched by a marking driver that will run the autocheck procedure in order to verify whether the exercise may be deployed.

Chapter 3

Submission protocol (the A protocol)

This chapter describes how to send one or more files to an acquisition server. The submission is an HTTP request that must be accompanied by two other pieces of information: *(i)* the identity of some user, *(ii)* the identity of the exercise that must be used to mark the job. The protocol conforms to REST style, it complies with HTTP rules.

The server first checks whether the exercise is valid (no job is accepted for URLs that do not mention a proposed exercise). Then, the server checks whether the student is valid. Finally, the format of the body of the HTTP request is analyzed; the size of the body must not exceed some predefined limits.

3.1 User information

The identity of the user, whose account will be debited, is specified via a cookie. This cookie is acquired by the submitting client (a browser or some other plugin plugged into some IDE) after registration and authentification on the authentication server `x.codegradx.org`. The cookie is named `u`, its value is a so-called "safe cookie" that is, some information safely identifying the user. A safe cookie is a regular cookie which payload (the identity of the user) is signed with the private key of the `x.codegradx.org` server, the concatenation of these two pieces of data (the identity of the user and the signature) is itself ciphered with the public key of the Web server. Therefore this cookie is normally unreadable (the identity of the user cannot be deciphered from the cookie) and unforgeable (only `x.codegradx.org` may create these safe cookies).

Here is an example of such a cookie. The cookie is sent along with the other HTTP parameters in the head of the request. It should be sent as a single line though the following example is cut to fit line length.

```
Cookie: u=UdfE3rNeMDlp9vLYAzFYyxzyOhhWYZ3buSoNH3a0H4Xws9mOUEiEEkg
  DJNSxkqtidT5_5f2taoOAo2PanzV2xFiOp1oesIsY9u7EW-upyf_8hUqtsKKgE1t0
  r__PbzPsPynYlxg77EsGld_C0kdIcZq3wmBOmcJn15wWZcNyjZCw@;path=/
```

3.2 Exercise information

The URL to post a file looks like `/exercise/`*E*`/job` where *E* identifies the exercise. The request should use the POST method. The sent file is somewhere in the body of the request (see next section). In case of success, the response code is 201 (created) and

a `Location` header tells which URL to use to monitor the job (this URL will address another server not the acquisition server whose only goal is to accept jobs).

An exercise is identified by a (long) string as in the following URL (cut into several lines to fit line length):

```
/exercise/UpxcmU_ca8rm6pBYKsk5hEcFVyP9j2s-gaqEim9hB0cE0V41G0G98z
T5WYKyxvrN3lQdiusXCdSBiVbXP4EKphBpKcCZqHPetp3TMmCcSY0WM4qnNaqqXs
2N37v6IqPSw/job
```

3.3 File content

There are two cases depending on the number of files to submit. If there is more than one file, they must be gathered in a tar gzipped archive file or in a zipped file. For instance, if you want to send the `a` and `b/c` files then the archive must contain these files with these names. Don't add superfluous prefix directory names (for instance, `d/a` and `d/b/c`). Normally your submitting client should take care of that point.

There are two ways to submit a single file. It may be sent as the whole body of the POST request. The MIME type should then be `application/octet-stream`. If you do not upload an archive file, you must send an accompanying `Content-Disposition` HTTP parameter to define the name of the uploaded file. This parameter is useless when uploading an archive file since the archive file already contains the name of the archived files.

Here is a minimal example of such an HTTP request that uploads a file named *solution.txt* containing two lines:

```
POST /exercise/UpxcmU_ca8rm6pBYKsk5hEcFVyP9j2s-gaqEim9hB0cE0V41G0G98z
  T5WYKyxvrN3lQdiusXCdSBiVbXP4EKphBpKcCZqHPetp3TMmCcSY0WM4qnNaqqXs
  2N37v6IqPSw/job HTTP/1.0
Cookie: u=UdfE3rNeMDlp9vLYAzFYyxzyOhhWYZ3buSoNH3a0H4Xws9mOUEiEEkg
  DJNSxkqtidT5_5f2taoOAo2PanzV2xFiOp1oesIsY9u7EW-upyf_8hUqtsKKgE1t0
  r__PbzPsPynYlxg77EsGld_C0kdIcZq3wmBOmcJn15wWZcNyjZCw@;path=/
Content-Type: application/octet-stream
Content-Disposition: inline; filename="solution.txt"

#! I'm the content
of the solution.txt file.
```

If you alternatively choose not to send the raw file as it stands, you may wrap it into an archive (say a tar-gzipped file) and POST the following HTTP request (where the content is displayed as backslashed bytes). Observe that the `Content-Disposition` is no longer required since the name of the file (*solution.txt*) is already known by the archive file. Care must be taken if alternate encoding (base64) or transfer (gzip) are used.

```
POST /exercise/UpxcmU_ca8rm6pBYKsk5hEcFVyP9j2s-gaqEim9hB0cE0V41G0G98z
  T5WYKyxvrN3lQdiusXCdSBiVbXP4EKphBpKcCZqHPetp3TMmCcSY0WM4qnNaqqXs
  2N37v6IqPSw/job HTTP/1.0
Cookie: u=UdfE3rNeMDlp9vLYAzFYyxzyOhhWYZ3buSoNH3a0H4Xws9mOUEiEEkg
  DJNSxkqtidT5_5f2taoOAo2PanzV2xFiOp1oesIsY9u7EW-upyf_8hUqtsKKgE1t0
  r__PbzPsPynYlxg77EsGld_C0kdIcZq3wmBOmcJn15wWZcNyjZCw@;path=/
Content-Type: application/octet-stream

\037\213\010\0...
```

The second way is to upload the file as part of the MIME type `multipart/form-data`. The name of the form field to use is `content`. Additionally if you do not upload an archive file, you must send an accompanying `Content-Disposition` HTTP parameter to

define the name of the uploaded file. This parameter is useless when uploading an archive file since the archive file already contains the name of the archived files.

Here is an example of such an HTTP request that sends a file named *solution.txt* containing two lines:

```
POST /exercise/UpxcmU_ca8rm6pBYKsk5hEcFVyP9j2s-gaqEim9hB0cE0V41G0G98z
  T5WYKyxvrN3lQdiusXCdSBiVbXP4EKphBpKcCZqHPetp3TMmCcSY0WM4qnNaqqXs
  2N37v6IqPSw/job HTTP/1.0
Content-Type: multipart/form-data; boundary=FW4EX
Content-Length: 168
Cookie: u=UdfE3rNeMDlp9vLYAzFYyxzyOhhWYZ3buSoNH3a0H4Xws9mOUEiEEkg
  DJNSxkqtidT5_5f2taoOAo2PanzV2xFiOp1oesIsY9u7EW-upyf_8hUqtsKKgE1t0
  r__PbzPsPynYlxg77EsGld_C0kdIcZq3wmBOmcJn15wWZcNyjZCw@;path=/

--FW4EX
Content-Disposition: form-data="content"; filename="solution.txt"
Content-Type: application/octet-stream

#! I'm the content
of the solution.txt file.
--FW4EX--
```

The previous examples use HTTP/1.0 but HTTP/1.1 protocol may also be used.

The acquisition server answers with a `jobSubmittedReport` XML document. This report (cf. Section 11) contains the URL to watch on a storage server where the `jobStudentReport` will appear.

3.3.1 Batch

Teachers may want to send a number of students' files in one go. For instance, they may collect files on students' computers, gather them then send them to the grading machine. This is a "batch". When submitted successfully a batch will return a location where the batch report will appear. That report will contain the URL of the grading reports for all the students' submissions.

The protocol is slightly more complex since, to ease processing results, the teacher may add some information to the batch and/or to the individual students' submissions.

First, the teacher has to collect the students' files to grade. Let say that these files are stored in a `students/` directory with one sub-directory per student. The content of the students sub-directories should be structurally similar as in:

```
% find students
1234567/options
5432176/options
7623451/options
```

The teacher may submit these sub-directories as tar-gzipped, zipped files or as they are. However, to ease further processing by the teacher, the teacher has to furnish some additional information in a `multiJobSubmission` XML document. Here is an example of such a `fw4ex.xml` file:

```
<?xml version="1.0" encoding="UTF-8" ?>
<fw4ex version="1.0">
 <multiJobSubmission label="{attempt: 1}">
  <job label="one"   filename="students/1234567/"/>
  <job label="two"   filename="students/5432176/"/>
  <job label="three" filename="students/7623451/"/>
 </multiJobSubmission>
</fw4ex>
```

In this XML document, the teacher tags the batch with a label (looking like a JSON record) then tags the three students' directories. These labels may help the teacher to associate the grading reports to the students. This is necessary since the grading machinery has no clue on who is graded. This information will be sent back to the teacher with the final batch grading report.

The batch is then tar-gzipped as follows:

```
% tar tzf batchToSend.tgz
./fw4ex.xml
./students/1234567/options
./students/5432176/options
./students/7623451/options
```

The `filename` attributes in the `multiJobSubmission` XML document tells where are the files within the tar-gzipped batch file. The `fw4ex.xml` must be located at the toplevel.

The tar-gzipped file (or zipped) is then POSTed to the URL of the exercise `/exercise/E/batch` with the methods exposed in Section 3.3.

The acquisition server answers with a `multiJobSubmittedReport` XML document. This report (cf. Section 11) contains the URL to watch on a storage server where the `multiJobStudentReport` will appear.

3.3.2 Client

To ease job submission, with client packages such as the Perl module `WWW::Mechanize`, is provided an URL that returns a form only appropriate to submit for the intended exercise:

```
/exercise/UpxcmU_ca8rm6pBYKsk5hEcFVyP9j2s-gaqEim9hB0cE0V41G0G98z
T5WYKyxvrN3lQdiusXCdSBiVbXP4EKphBpKcCZqHPetp3TMmCcSY0WM4qnNaqqXs
2N37v6IqPSw/form
```

For instance, a submitting client, in Perl, may be as short as:

```
use WWW::Mechanize;
my $mech = WWW::Mechanize->new;
$mech->cookie_jar({ file => $cookieJarFile }); # set the student
my $url = $serverName . "/exercise/$exerciseId/form";
$mech->get($url);                              # get the form
$mech->submit_form(
 form_name => 'form',                          # fill the form
 fields => { 'content' => $nameOfTheFileToSend },
 button => 'do',                               # post the file
);
```

To ease batch submission, an alternate form exists.

3.4 Errors

In case of errors or problems, the response has an HTTP erroneous code (from the 400 family) and is associated to a string starting with `FW4EX` followed by an error code (a three figure integer prefixed with the letter *e* and suffixed with a human readable message.

3.5 Summary

An acquisition server is not browsable. One may only perform the following public actions with the associated URL (where E identifies the exercise and J is the UUID identifying a job or a batch):

`GET /alive`	return a JSON document telling whether the server is alive
`GET /jobload`	return a JSON document with the load that is, the number of jobs waiting to be marked.
`GET /exercise/E/state`	check whether an exercise is ready to accept job
`GET /exercise/E/form`	get an XHTML form to post a job towards an available exercise
`POST /exercise/E/job`	post job towards ready exercises
`GET /job/J/state`	check whether a job was accepted
`GET /exercise/E/batchform`	get an XHTML form to post a batch towards an available exercise
`POST /exercise/E/batch`	post batch towards a ready exercise
`GET /batch/J/state`	check whether a batch was accepted

/alive returns a JSON document with date and version of the server. This allows to check whether the A server is alive.

/jobload returns a JSON record showing the load of the acquisition server. The record lists the number of archived jobs not yet fetched by the marker, the date of the oldest job waiting to be marked (if any) and the time (the (Un*x) number of seconds since the epoch) when was produced this record. This record is refreshed every minute.

/exercise/E/state checks whether an exercise is ready to accept jobs (E identifies the exercise). If the exercise exists and is available, the HTTP return code is 200. The body of the HTTP response contains the `FW4EX e000` message meaning that the exercise is ready.

/exercise/E/form returns an XHTML form to post a job towards an available exercise (E identifies the exercise). This URL is not very useful.

/exercise/E/job posts job towards ready exercises (E identifies the exercise). If the job is correctly stored on the acquisition server, the HTTP return code is 201 (Created) and is accompanied with a `Location` parameter with the URL to monitor.

/job/J/state checks whether a job was accepted (J identifies the job created by the previous URL). If such a job exists on the acquisition server, the HTTP return code is 200. The body of the HTTP response contains the `FW4EX e001` message meaning that the exercise is archived. Pay attention that this request may only be answered for a short while. Once the grading report is safely stored on the storage server, the job will be removed and the URL will return an error. The URL to use to obtain the grading process is the URL returned by the POST request.

/exercise/E/batchform returns an XHTML form to post a batch towards an available exercise (E identifies the exercise).

/exercise/E/batch posts batch towards ready exercises (E identifies the exercise). If the batch is correctly stored on the acquisition server, the HTTP return code is 201 (Created) and is accompanied with a `Location` parameter with the URL to monitor.

/batch/J/state checks whether a batch was accepted (J identifies the batch created by the previous URL). If such a batch exists on the acquisition server, the HTTP return code is 200. The body of the HTTP response contains the `FW4EX e001` message meaning that the exercise is archived. Pay attention that this request may only be answered for a short while. Once the grading report is safely stored on the storage server, the batch will be removed and the URL will return an error. The URL to use to obtain the grading report is the URL returned by the POST request.

Responses normally are in XML. HTTP error codes should not be ignored.

When returning JSON data, JSONP (or JSOD *Javascript on demand*) mode is also supported. The appropriate query string parameter may be named `cb` or `callback`.

3.6 Administrator supplementary interface

Other requests to the acquisition server are possible but they are restricted to administrators. In particular, the acquisition server is regularly polled by the grading driver daemon in order to get new jobs to mark. Here are the administrator requests (where J (an UUID) identifies a job):

`GET /jobs`	returns the list of pending jobs (identified by their UUID)
`GET /job/J/submission`	returns the J job
`DELETE /job/J`	deletes the J job
`GET /batches`	returns the list of pending batches (identified by their UUID)
`GET /batch/J/submission`	returns the J batch
`DELETE /batch/J`	deletes the J batch
`GET /checkexercise/E`	decrypt part of the exercise identifier
`GET /checkcookie`	decrypt part of the user's cookie

/jobs This request returns the list of pending jobs. They are sorted by creation date (oldest first). The body of the answer is an `acquisitionServerState` XML document (see `fw4ex.rnc` grammar in chapter 11).

/job/J/submission This request returns a job identified by J (an UUID). A job is a compressed archive (tar gzipped) containing a `fw4ex.xml` descriptor describing the job (the user, the intended exercise, the creation date) as well as the files contained in the job. The `fw4ex.xml` descriptor is a `jobSubmission` XML document (see `fw4ex.rnc` grammar in chapter 11).

/job/J This request deletes a job J from the pending jobs that are archived on the acquisition server.

/batches This request returns the list of pending batches. They are sorted by creation date (oldest first). The body of the answer is an `acquisitionServerState` XML document (see `fw4ex.rnc` grammar in chapter 11).

/batch/J/submission This request returns a batch identified by *J* (an UUID). A job is a compressed archive (tar gzipped) containing a `fw4ex.xml` descriptor describing the batch (the user, the intended exercise, the creation date) and the name of the related jobs. The `fw4ex.xml` descriptor is a `batchSubmission` XML document (see `fw4ex.rnc` grammar in chapter 11).

/batch/J This request deletes a batch *J* from the pending batches that are archived on the acquisition server.

/checkexercise/E This request checks the identifier E and returns (as a text/plain document) the UUID that identifies an exercise. This is a technical URL not very useful for clients.

/checkcookie This request checks the user cookie and returns (as a text/plain document) the number that identifies a user. This is a technical URL not very useful for clients.

Chapter 4

The XML server (the X protocol)

This chapter describes some REST-based protocols provided by the `x.codegradx.org`:

1. This server provides a way to be authenticated and thus delivers cookies required to interact with other servers.

2. This server also provides information on the user and his history of interactions with the constellation.

3. This server provides administrative methods to inspect the database and the state of the constellation.

4.1 User authentication

When addressed with a safe cookie identifying a user, the `x` returns an *authenticationAnswer* that is, an XML document describing the user. This URL allows to check whether a cookie is valid.

There might be more than one way to get a valid safe cookie. To post a *login* and a *password* to the `/direct/` URL is a possibility. The *login* and *password* are checked in the database and if they match a registered user, a safe cookie is generated and returned accompanied with an *authenticationAnswer* XML document describing the user.

4.2 Summary

The XML server provides the following URLs.

`GET /`	Describe the current authenticated user with an XML *authenticationAnswer* or JSON record. Otherwise return the default banner of the server.
`GET /alive`	return a JSON document telling whether the server is alive.
`GET /dbalive`	return a JSON document telling whether the server is alive and has a working access to the database.
`POST /direct/`	Should be accompanied with a *login* and a *password*. If they are correct, this service returns a safe cookie and an *authenticationAnswer* XML report or JSON record.
`GET /direct/checkForm`	Return an XHTML form allowing an user to submit *login* and *password* information. This is an helper service. This URL is not very useful.

The format of the answer (XML or JSON) is chosen after the *Accept* parameter. When returning JSON data, JSONP (or JSOD *Javascript on demand*) mode is also supported. The appropriate query string parameter may be named `cb` or `callback`.

4.3 Personal supplementary interface

These services may be freely invoked by the requester as far as the returned information concerns him. Therefore, one may obtain his personal information (jobs, badges, invoices, etc.) but teachers may obtain information about their students and exercise authors may obtain information about the jobs graded against the exercises they create. Above all, admins may see everything.

`GET /person2`	get the history that is, the list of all graded jobs submitted by the requester that is, a *studentHistory* XML document.
`POST /person/selfmodify`	modifies the information associated to the requester. One may thus change one's own password, pseudo or email. The result is a *authenticationAnswer* XML document.
`GET /campaigns/`	lists the campaigns associated to the requester whether as a student or as a teacher. The result is a JSON record, campaigns have a name, a start time and an end time. This allows to know which are still active.
`GET /campaign/C`	Describes the campaign by its name C. The result is a JSON record with the list of all exercises (name, nickname and uuid only).
`GET /skill/C`	Request the skills of the various (anonymous) participants of a campaign. The own skill of the requester is also provided so one may judge of its own skill relatively to the others. The parameter C is the name of the campaign (an identifier often suffixed by the name of a term).
`GET /history/campaign/C`	Lists the jobs of the requester associated to the exercises of the campaign C. Return a JSON result.
`GET /jobs/person/P`	Request the jobs submitted by person P where P is the id of a person. The list is controlled by three query parameters `offset`, `count` and `after`. The output format depends on the `Accept` parameter and may be `text/xml` or `application/json` or `text/csv`.
`GET /jobs/job/J`	Request the jobs identified as J where J is the uuid of the job (there is one per Marking Driver that graded the job). The list is controlled by three query parameters `offset`, `count` and `after`. The output format depends on the

The `count` query parameter tells how many results are required. By default, `count` is limited to 20. With the `offset` parameter, you may thus require the `count` results numbered `offset`, `offset+1`, `offset+2`, ..., `offset+count-1`. Offsets start at 0.

The last query parameter, `after`: a date, filters results to be more recent than that date. Dates are expressed as three numbers: *year-month-day*.

4.4 Administrator supplementary interface TO BE FINISHED

The following services require authentication and may only be used by administrators.

`POST /person/create`	registers a new person and login. The following information must be given: login, lastname, firstname and email. Additionally, password and pseudo may also be given. The returned document is a authenticationReport XML document.
`POST /person/modify/LOGIN`	modifies the information associated to a person and/or associated login. One may change his password, pseudo or email. The returned document is a authenticationReport XML document.

Chapter 5

Dynamic storage protocol (the D protocol)

Usually there are more than one running grading server, a D server is co-located with an S (storage) server; for every job, it receives the grading reports produced by the running grading servers. It compares grading reports and keeps the best one in case of discrepencies between grading servers. The most usual case is a grading server with an internal booting error yielding a mark equal to zero for every graded job. Of course, the administrator of the constellation is notified!

5.1 Summary

The dynamic storage server provides the following URLs.

`GET /alive`	return a JSON document telling whether the server is alive
`PUT /storer/J`	Store a file into the directories served by an accompanying S server.
`POST /storer/J`	Store a file into the directories served by an accompanying S server.

In fact, a dynamic storage server stores job reports, exercise reports, batch reports, invoices, badges, etc. This server is only used internally from marking drivers or various other management processes.

Chapter 6

Storage protocol (the S protocol)

There is no such protocol. Storage servers are addressed with regular GET URLs looking like:

```
/s/U/U/.../I/D/uuid.xml       Grading report  - jobStudentReport
/s/U/U/.../I/D/uuid_.xml      Author report   - jobAuthorReport
/e/U/U/.../I/D/uuid.xml       Exercise report - exerciseAuthorReport
/b/U/U/.../I/D/uuid.xml       Batch report    - multiJobStudentReport
/c/U/U/.../I/D/campaign.pdf   Certificate
/f/U/U/.../I/D/f*.pdf         Invoice
```

`U/U/.../I/D` is the UUID converted into a series of directories with names reduced to a single letter.

Chapter 7

Case studies

In this chapter, some use cases of CodeGradX are described. This is an area where much progress were made since 2008. The trend now is to hide the various HTTP protocols with some JavaScript libraries, see 10.4.1.

7.1 The MOOC "Socle informatique" from CNAM

In 2017, from april to july, the CNAM had been using the CodeGradX infrastructure to grade C programs proposed in a MOOC around the C programming language. A new organization was set up to ensure robustness and speed.

First, the central database was moved to an AWS (Amazon Web Service) RDS (Relation Data Server) machine accompanied by a backup server. Then a set of one A, E, X and CC servers were set up on an AWS ECS (EC2 Container Service) machine while a Marking Driver and one marking slave were deployed on another AWS ECS machine. The CC server was the dedicated Web server displaying the exercises of the MOOC.

The A, E, X and CC servers were named `a2.codegradx.org`, `e2.codegradx.org`, `x2.codegradx.org` and `cc2.codegradx.org`. The configuration of cc2 was set up to prefer addressing a2, e2 and x2 (which are all co-located on the same physical machine). The e2 server was configured to statically hold all the exercises required for the MOOC.

The marking driver and its slave was only polling the a2 and e2 servers every 5 seconds (instead of the usual 10 seconds).

Technically, the two ECS machines were running Docker containers. The machine hosting a2, e2, x2 and cc2 hosts an additional nginx container to proxy the http and https ports towards the appropriate container.

The MOOC was hosted by FUN (France Université Numérique) running Open EdX, exercises were proposed as buttons leading the student towards CodeGradX. The first access to CodeGradX registers the student with EdX provided information (pseudo and email). Other accesses display the exercise directly. The final mark is sent back to EdX using the LTI protocol (Learning Tools Interoperability).

7.1.1 LTI Protocol

Within EdX, you must define first the LTI provider with an LTI passport. Then you define an LTI consumer with an host (here `https://x2.codegradx.org/fromedx/`) and custom parameters such as:

```
[ "site=cc2.codegradx.org",
  "uriprefix=/directoneliner",
  "exercisename=cnam.mooc.socle.convert.1",
  "lang=fr",
```

```
    "groups=cnam-socle-informatique-2017s1"
  ]
```

The URL of the exercise is derived from `site`, `uriprefix` and `exercisename` custom parameters. The X server will redirect the student towards this URL after authentication and registration of the students. This is possible since EdX authenticates users, EdX can be trusted and the LTI protocol is trustful. The `site` is the Web server that displays the set of exercises to students.

The `lang` can be `fr` or `en`.

The `groups` (optional) custom parameter lists (comma-separated) group names. Students are automatically included in these groups. Groups are used to list students and compute statistics.

7.1.2 Evolutions

Some machines are now running on Google Computing Platform. This is the case of a couple of one Marking Driver and its Marking Slave. This machine also hosts a load balancer thus providing `a.codegradx.org` distributing work towards `a3.codegradx.org`, `a4.codegradx.org` and `a5.codegradx.org` and similarly for `e.codegradx.org` and `x.codegradx.org`.

Due to the characteristics of the various machines: some are fast with a small disk (25G) while some others are slower but with a big disk (2T); storage servers have more individual roles. The fastest machine holds job reports only for a day, they are then migrated to a slower but bigger server. After 3 months, job reports are migrated to a slower storage server alike Amazon S3.

Chapter 8

Authors guide

This chapter suggests how to create an exercise. It is useful for authors that may use some libraries to ease their job. Once an exercise is built, authors should look at the CodeGradX agent (page 105) to automatize the test of their exercise.

8.1 Choose an identifier

By convention, an exercise is developped in a directory whose name is the unique name that will identify your exercise now and for ever therefore a name cannot be reused.

Usually the name is similar to a Java package name such as `tld.yourdomain.thema.nickname.version`. Some prefixes are reserved and you cannot use them; these are `com.paracamplus`, `org.paracamplus` and `org.fw4ex`. However, the `org.example.` prefix is freely usable by every author to test the infrastructure. There is no guarantee of permanence though for exercises named with this prefix.

All authors are associated with at least one prefix by default `lastname.firstname`.

Once a name is chosen, say `queinnec.christian.essai.1`, create a directory of that name and create a file name `fw4ex.xml` within it. Since this file must be valid, download the grammar for CodeGradX in order to check your `fw4ex.xml` with:

```
xmllint --relaxng fw4ex.rng name.queinnec.essai.1/fw4ex.xml
```

Of course, you may also download one of the public exercises and clone its `fw4ex.xml` file.

The rest of this section details some of the more interesting XML section you, as an author, has to complete in the `fw4ex.xml` file. For more precise information, see the commented grammar.

The name you chose for the exercise must be recorded in the `identification` section. The nickname is a short name, usually a single word, that identify the exercise among a series of exercises. The `summary` is one sentence that roughly defines what the exercise is all about. The `description` is a longer text that describes in more details what is the exercise, what it requires, what it checks. This is not as detailed of course as the stem which may appear in the `introduction` section. The identification, summary and description are public information displayed in various forms. The other parts of the exercise such as stems are not public.

The cost is what you may want to be paid when one student runs your exercise. It cannot be less than 0.01€ and more precisely less than a fixed price computed from the time it takes to grade a student's submission. Currently, the infrastructure requires no fee to be used and therefore pays nothing. Exercises are free software!

The `equipment` section shows which files should be available for students to practice the exercise.

Questions are specified in the `question` section. It contains an `expectation` section that specifies which files are expected from the student. If these files are not present, then the student will be informed and no grading script will be run. The `stem` section describes the stem of the question. As usual, the text looks like XHTML but with less tags, look at the commented grammar for accurate details.

The `autochecking` section specifies where are the pseudo-copies and which mark they should grade, see next Section for more details.

The `grading` section is where you specify the scripts to run in order to grade a student's submission. You may have a number of scripts per question, you may also have scripts out of questions. Quite often an initial script my fix student's files before entering the scripts associated to the first question.

Sometimes to grade a student's submission requires some libraries to be available. The `initializing` section allows resources of an exercise to be prepared when the exercise is deployed so well ahead before grading submissions. Compiling C or Java code that does not depend on the copy of the students should be done in the `initializing` section.

There are tons of other options possible in this `fw4ex.xml` file, read the commented grammar for more information.

8.2 Naming exercises

More than one name may qualify an exercise:

the long name identifies uniquely an exercise. It starts with one of the allowed prefixes an author have. It should be a prononceable name often looking like `fr.upmc.course.topic.` or `cnam.mooc.socle.` or `fr.queinnec.christian.` that is a series of words separated by dots. Quite often the words are ordered from the most general to the most specific. The long name is used by authors.

Some prefixes are reserved such as `com.paracamplus` or `org.codegradx`. Some prefixes such as `org.example.` or `tmp.` are public and may be used and abused (that is, overwritten) by anybody.

the nickname or short name is a single word that identifies an exercise within a collection of exercises. This name is often used when displaying information to students, for instance, a list of exercises to be done or an history of exercises' results. Quite often, the nickname is the last non numeric component of the long name.

the internal UUID qualifies a tgz that is a precise instance of an exercise. This UUID is assigned to the exercise.tgz by the grading engine. This is the name used to record grading reports in the database. An internal UUID is never re-used.

Students only have access to exercises via safe cookies, a safe cookie contains the UUID of an exercise and not its long name.

An author can submit a new exercise.tgz with an already used long name. If the exercise is successfully autochecked, it will get a fresh UUID and be available via its own new safe cookie. At the end, a long name may be associated to more than one exercise.tgz. If you want your students to use the new instance, you have to update the safe cookie that led to the former instance in all the places where you mentioned the long name. Usually there is a single landing page displaying the collection of exercises to practice, this is the page to update.

Pay attention when re-using a long name since students are aware of short names, they may take notice of long names but will probably never read those unreadable UUIDs. Updating the stem of an exercise is not a problem, neither is adding new pseudo-copies. However improving the grading scripts should not alter the total mark and in the case where the new grading scripts change the marks of former jobs, it must

not lower these old marks or you might have problems with the students who do not understand why their mark get lowered!

If you alter significatively the grading scripts, you should probably change the long name (this is why long names often finish with a version number as in `org.codegradx.js.min3.2`) and update the safe cookie that led to the former instance in all the places where you mentioned the previous long name.

8.2.1 Naming new exercises

When an author submits a new exercise.tgz, the fresh exercise will be checked. A new UUID is given to the exercise, the `fw4ex.xml` manifest file is checked, the pseudo-jobs are marked and their resulting mark is compared to the expected mark. During these checks, the exercise has a temporary long name prefixed with `autochecked.` prefix, after decoding the manifest, its long name is prepended with a `tmp.` prefix. If everything went well, the exercise is recorded in the central database with a `tmp.` prefix.

In the central database, some additional verifications are performed:

- The long name of the exercise must begin with one of the author's authorized prefixes.
- If a former exercise with the same claimed long name is already present then only the owner of the exercise of that name can reuse that name.

If any of these verifications fail, the `tmp.` prefix is not stripped and the first authorized author's prefix is prepended to the long name.

Pay attention when a long name is associated to more than one exercise.tgz. Only the UUID of these exercises differ and the most recent one is, quite often, choosed by default.

8.3 Overall directory structure

An exercise is a tar gzipped file containing everything necessary for the student to practice the exercise and for the grading server to mark the files submitted by students. It also contains non-regression tests to check whether the grading machine is able to perform grading that is, it contains all the required or well-configured resources (programs, compilers, validators, grammars, data files, dictionaries, configurations, etc.) needed to grade.

In favor of convention instead of configuration, it is strongly suggested that this directory is structured as follows:

```
./fw4ex.xml
./data/
./tests/check.sh
./tests/1.data
./tests/2.data
./pseudos/null/
./pseudos/perfect/solution
./pseudos/half/solution
```

The `data/` directory contains the files to ship to the student. These files are needed to practice the exercise. This directory may not exist if no such files are needed. Files that must be available by students must be declared in the `equipment` section in the `fw4ex.xml` file. Here is an example of an `equipment` fragment where the `data/lib.sh` file is given to the students:

```
<equipment>
  <directory basename='data'>
    <file basename='lib.sh'/>
  </directory>
</equipment>
```

The `tests/` directory contains the files required to grade a job (or a pseudo-job). It usually contains some shell (or other language) scripts to drive the grading and data files that the student's file have to process. The grading scripts may also directly appear inlined within the `fw4ex.xml` descriptor.

The `pseudos/` directory contains the pseudo-jobs to ensure non-regression. It is again strongly suggested to have at least three pseudo-jobs (but at least two are mandatory):

`null` this pseudo-job contains nothing: its final mark should therefore be zero. If the final mark is not zero then a student that does nothing already gains something!

`perfect` this pseudo-job contains the solution that is the required files behaving as specified in the stem. The final mark should be equal to the total mark specified in the `fw4ex.xml` description. If the final mark is not the expected one then it might be the case that the grading machine is lacking some programs you need. Check the report and complain to CodeGradX maintaineers!

`half` this pseudo-job contains incorrect answers and should get an intermediate grade (nor zero, nor the total mark). This pseudo-job is important as it avoids two common pitfalls:

1. a bug in the grader may give the maximal mark to every non empty job. This bug is not detected by the previous pseudo-jobs. This is particularly the case if you compare answers to the perfect one. If the perfect solution fails to produce the correct answer then, to compare it to itself always succeed and always give the maximal mark!
2. it is advised to avoid all-or-nothing exercises. For instance, asking for a program that only returns YES or NO is not advised since a program that always return a constant answer will probably succeed half of the time and get half of the maximal grade without much effort.

I usually add in the `pseudos/` directory some students' solutions that trigger bugs or weird behaviours in the grader. This allows to make the exercise more robust as time passes by.

Finally, the `fw4ex.xml` descriptor ties everything. It contains (or references where is) the stem, the expected marks for the pseudo-jobs, etc.

8.4 Grading

There are at least two ways to grade. Grading may be done by comparison to a correct solution. It may also be done by inspection of the result.

8.4.1 Grading by comparison

Grading by comparison is a process that usually passes through the following steps:

1. **Check expectations** — Verify that the files expected from the student are present, non empty, executable, etc. The sole presence of files may be specified in the `fw4ex.xml` descriptor.

2. **Feedback** — Make student's files appear in the grading report. This is useful for the student since he may check that these are really the files he submitted. It is also useful for the author of the exercice since the report is self-contained.

3. **Loop** — For all test cases (and it is better to know how many of them there are):

   (a) **Setup** — Set up the test that is: populate then jump to a directory, uncompress some data files, etc.

   (b) **RunStudent** — Run the student's programs

   (c) **ShowStudentAnswer** — Show the results of the student's programs

   (d) **NormalizeStudentAnswer** — Normalize the results that is remove superfluous spaces or newlines, etc.

   (e) **Compare** — Gauge the normalized result. There are at least two possibilities: an oracle or a comparison to a correct solution. Using a comparison is often done as follows:

      i. **RunAuthor** – Run the author's programs

      ii. **ShowAuthorAnswer** — Show the results of the author's program.

      iii. **NormalizeAuthorAnswer** — Normalize the results as before.

   (f) **EvaluateGain** — Determine the final grade for this test case. This phase may be disseminated through the previous phases but it is cleaner to separate this phases so the ponderation may evolve independently.

To help authors, some libraries exist to factorize common tasks. These are *bash* libraries that should be `source`-d. Even more, a set of configurable patterns are provided for grading reports. Depending on the kind of exercises you want to design, the programming languages used, you may start with one of the following patterns and tailor it to your needs.

8.4.2 Grading by inspection

Grading by inspection is almost similar. In the above enumeration the **Compare** step is performed by a dedicated script that checks whether some properties hold or not.

8.4.3 Marks

Usually, I mark between 0 and 1. This is particularly useful if you want to normalize the marks of a series of exercises. However, I often display marks with a markFactor of 100 since students prefer getting 100/100 rather than 1/1!

If you have only one exercise for instance an examination, marks can go up to 72 or whatever highest mark is convenient.

8.5 Author's script

An author script follows very simple convention:

- The stdout will be sent to the student, the stderr will be sent to the author (it may also be read by CodeGradX maintaineers).

- The stdout must be a valid XML document so pay attention to close your tags (even when scripts are abruptly terminated!). The CodeGradX platform tries to tidy this XML but tidying cannot fix all problems!

- The script runs in the HOME directory of a student under the identity of this student. Therefore it may not alter the files owned by the author.
- The script must take great care of the student's files since it may modify them (at least, it has the right to). It is safer to avoid, where possible, any modification of the HOME directory. Use TMPDIR for temporary files (don't use `/tmp` directly as it is not safe). The CodeGradX platform creates one TMPDIR per job so every job has its own unshared space.
- Always confine the student's programs! Remember that author's programs are also confined so what you give to the student (CPU, streams) is taken from your share.
- Always pass your data through `transcodeCarefully` to avoid generating bad XML.

8.6 Environment

There are a number of variables that may be used by authors' scripts. Except for `TMPDIR` all the other variables, prefixed by `FW4EX_` should be absolutely hidden from student's programs (this is normally ensured by the use of the `fw4ex_confine` library function).

`TMPDIR` This variable specifies the directory to use for temporary files. Don't use `/tmp`, you may not have the right to! This temporary directory will be removed at the end of the job but you may use it during the whole job therefore you may store information in it from one question to the other.

`FW4EX_LIB_DIR` This variable contains the name of the directory that contains the shell library for authors. Most often, an author's script starts with:

```
source $FW4EX_LIB_DIR/basicLib.sh
source $FW4EX_LIB_DIR/moreLib.sh
```

`FW4EX_BIN_DIR` This variable contains the name of the directory that contains some utilities for authors though most of these utilities are better used via shell functions from the `basicLib.sh` library.

`FW4EX_EXERCISE_DIR` This variable contains the name of the directory that contains the inflated exercise. This directory contains the `fw4ex.xml` file and all the other files packed in the exercise tar gzipped file.

`FW4EX_JOB_ID` This variable contains the name of the current job. This variable is useless for authors and is only needed by the `fw4ex_win` command.

`FW4EX_QUESTION_NAME` This variable contains the name of the current question (if the script is embedded within a `question` element, of course).

Additional variables may be pathnames to other resources, for instance, for Java (see Section 8.11.1):

`FW4EX_JUNIT_JAR` is the path leading to the currently installed JUnit jar (currently 4.11). This JUnit is wrapped with JCommander 1.27 (from Cedric Beust) and some additional TestRunner making easier to mark with JUnit.

`FW4EX_JAVA_BIN` is the path leading to some helper classes.

8.7 Utilities

There are two complex utilities: `win` and `confine` and a simpler one: `transcodeCarefully` upon which rest many library functions (to be seen in the next sections). These utilities are often used indirectly via the `fw4ex_win`, `fw4ex_confine` and `fw4ex_transcodeCarefully` library functions.

8.7.1 win

`win - produce a mark`

SYNOPSIS

A single argument is a formula to evaluate:

```
win.pl 1/2                                      # yields 0.5
win.pl 1.3/3                                    # yields 0.43
```

More than one argument is a parameterized shape followed by a number. Here the shape is triangular:

```
win.pl triangular 0 10 50 100 -- 0              # yields 0
win.pl triangular 0 10 50 1000 -- 5             # yields 500
win.pl triangular 0 10 50 1000 -- 10            # yields 1000
win.pl triangular 0 10 50 1000 -- 30            # yields 500
win.pl triangular 0 10 50 1000 -- 50            # yields 0
```

Formulas may be used everywhere:

```
win.pl triangular 1-1 100/10 100/2 '10*100' -- '7*11'    # yields 0
```

DESCRIPTION

This program takes some arguments and produces a mark onto its standard output. It basically takes the description of a function (say `f`) followed by a number (say `x`) then computes and emits `f(x)`.

The emitted mark is always limited to at most two decimals.

Some shapes are predefined. They are described below.

SHAPE equal

The `equal` shape is a kind of finite Dirac. The command looks like:

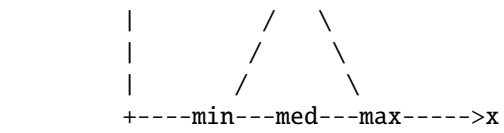
```
win.pl equal xmed ymax -- x
```

If `x` is equal to `xmed` then the `ymax` mark is emitted otherwise the won mark is `0`.

SHAPE triangular

The `triangular` shape looks like:

```
  y
  ^
  |
ymax          +
  |          / \
```



The command looks like:

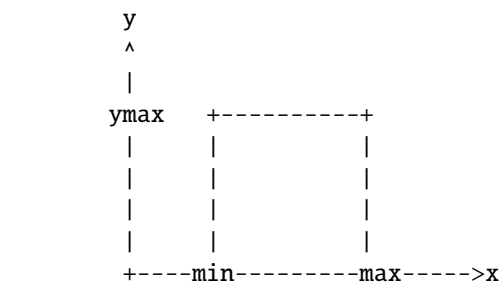
```
win.pl triangular xmin xmed xmax ymax -- x
```

The maximal mark is `ymax` and is won when `x` is equal to `xmed`. Out of the interval [`xmin`, `xmax`], the emitted mark is `0`. Otherwise the won mark is linearly interpolated.

One may parameterize the shape with `xmin = xmed` or `xmed = xmax`. Use the shape `equal` when `xmin = xmed = xmax`.

SHAPE rectangular

The `rectangular` shape looks like:



The command looks like:

```
win.pl rectangular xmin xmax ymax -- x
```

The maximal mark is `ymax` and is won when `x` is between `xmin` and `xmax`. Out of the interval [`xmin`, `xmax`], the emitted mark is `0`.

8.7.2 confine

The `confine` program runs a program in a restricted context with few if any rights. It is possible to limit the duration, the number of emitted characters on the stdout or stderr, to mask variables and so on. See also the `fw4ex_confine` library function from the `basicLib.sh` library, see 8.8.2 for further details.

Some terse documentation may be obtained with the `--help` option:

The numerous error codes may be obtained with the `--error-codes` option:

```
confine 2013Mar15 09:56
FW4EX error codes:
  209 Unknown option:
  210 Missing program name!
* 211 Failed exec()!
  212 Missing equal sign in environment!
  213 Failed setsid()!
  214 Missing number in option!
  215 Extraneous characters after number in option!
  216 Received unknown signal!
  217 Confined process's unknown exit value
  218 Failed sigaction() - SIGXXX
```

```
  219 Failed setitimer()!
  220 Failed waitpid()!
  221 Failed fork()!
* 222 Confined process did not exit by itself!
  223 Failed pipe() - stdXXX
  224 Failed dup2() - child stdXXX
  225 Failed close() - child stdXXX
  226 Failed read()!
  227 Failed write()!
* 228 Too much output!
  229 Weird child death!
  230 Failed malloc()!
  231 Failed fopen() - for XXX!
  232 Failed unsetenv()!
  233 Failed reset groups

The exit value of this program may be:
  211 if the program cannot be started with exec().
  222 if the confined program was killed because it exceeds its specified
      duration
  228 if the confined program was killed because it exceeds the number
      of characters to be output on its stdout or stderr streams.
  the exit value of the confined process if it terminates naturally.

Except 211, 222 and 228, exit values in the range 209-232 are internal
errors that normally should not be seen!
```

The `confine` command takes a lot of options followed by a `--` option followed by the command to be confined. Here are some simple uses to illustrate some possible confinements:

```
% confine --exit-file errcode -- date --rfc-2822
Thu, 18 Dec 2008 20:34:09 +0100
% cat errcode
0%
% date; confine --cpu=1 --exit-file errcode -- sleep 10 ; date ; cat errcode
Thu Dec 18 21:04:16 CET 2008
Thu Dec 18 21:04:17 CET 2008
222
% confine --maxout=32 --exit-file errcode -- yes coucou ; cat errcode
Thu Dec 18 21:05:14 CET 2008
coucou
coucou
coucou
coucou
couc228
% confine --maxout=32 --exit-file errcode -- laskdjfa ; cat errcode
211%
```

The `errcode` file contains the exit code of the confined command. An exit value produced by the confined program is not followed by a newline. An exit value produced by the confiner itself (i.e., the `confine` command) is followed by a newline.

8.7.3 transcodeCarefully

```
transcodeCarefully - sanitize a stream of chars into XML/UTF8
```

SYNOPSIS

```
transcodeCarefully.pl < someFile
cat someFile | transcodeCarefully.pl --line
```

DESCRIPTION

This program processes its standard input and convert it into a stream of characters suitable to be embedded in an XML UTF-8-encoded document. Quite often normal characters just pass through. Non-printable characters are converted into flagged printable characters. XML special characters are converted into the named equivalent entities. These special characters are the apostrophe, the double quote, the less than and greater than signs.

There is an option `-lineNumber` that numbers the lines. The numbering style is currently not configurable. Line numbers are wrapped inside a `lineNumber` element so it may be styled with CSS style sheets.

Another option is the total number of incorrect UTF-8 bytes that can be accomodated. More than this number (100 by default), these incorrect bytes no longer appear in the output. If the limit is reached, the package exits with a bad error code.

The input is expected to be UTF-8 encoded. If some bytes are encountered that do not respect UTF-8 they are emitted in hexadecimal with a leading funny Unicode character.

There is no attempt to check whether the swallowed or emitted XML is well-formed or valid.

8.7.4 headtail.sh

This utility reads its input stream and outputs only the N first lines and the N last lines. By default, N is 5. For instance,

```
% for letter in a b c d e; do echo $letter ; done | headtail.sh -1
a
e
```

8.8 Libraries

This section explains the various libraries, it focuses on the main utilities. Some libraries (in the `Languages` directory) are tailored for specific programming languages. Other libraries (in the `Patterns` directory) correspond to usual cases when testing programs. Quite often, an author sources the `basicLib.sh` library then the `moreLib.sh` library. The other ones are only sourced if required.

These author libraries are available online. They are useful for authors if they want to test their exercises before submitting to CodeGradX.

8.8.1 Local checks

In order to check grading scripts on a development machine before submitting an exercise to the CodeGradX infrastructure, download the author libraries in some directory, say `/tmp/authorlib` then if we suppose that the current directory contains the manifest i.e., the `fw4ex.xml` file, configure the following environment variables:

```
export FW4EX_EXERCISE_DIR=$(pwd)
export FW4EX_LIB_DIR=/tmp/authorlib
export FW4EX_BIN_DIR=/dev/null
```

```
export FW4EX_JOB_ID='abcdefghijklmnopqrstuvwxyz'
export TMPDIR=/tmp/mygradingresults
mkdir -p $TMPDIR
```

The grading script must source the `cxshim.sh` library, that library provides light and unsafe equivalents of the binary executables `confine` and `transcode`.

And run your grading script, say `mygradingscript.sh`, for the `perfect` pseudo-job. The job report will appear on the standard output.

```
cd pseudos/perfect
$FW4EX_EXERCISE_DIR/mygradingscript.sh
```

You may alternatively use the two next template scripts to ease authors' tasks. These template scripts are stored in the `Dev` directory of the author libraries. You may need to edit them to suit your configuration.

localEval.sh This script does not need a running VMauthor virtual machine. It only uses shell capabilities and you may edit it to suit your specific needs. As above, we suppose that `test/check.sh` is the grading script, we also suppose that the current directory contains the pseudo submissions in the `pseudos` directory. You may get the job report for one pseudo job, say `half`. with:

```
./localEval.sh -p half
```

You may check all pseudo submissions that is all the subdirectories of the `pseudos` directory with the `-a` option:

```
./localEval.sh -a
```

The associated job report will be produced on the standard output. If your grading script is not named `test/check.sh` then you may specify its real name with

```
./localEval.sh -p half -s mygradingscript.sh
```

Remember that the `confine` and `transcode` binary executable are only simulated (if your grading script source the `FW4EX_LIB_DIR/cxshim.sh` library) so duration and output are not controlled at all: you may shoot yourself in the foot!

VMauthorEval.sh This template script checks an exercise with a running VM for authors, see 10.5. It will check the manifest, recreate the tgz, submit it and wait for results. All results are stored in a freshly created directory.

The script needs to know where is the `codegradxvmauthor.js` script see 10.4, the grammar `fw4ex.rng` and the IP number of the VMauthor virtual machine. You may edit the template with those settings or specify them as environment variables.

```
GRAMMAR=../Grammars/fw4ex.rng ./VMauthorEval.sh
```

If the exercise can be deployed, a successful final message will appear.

8.8.2 basicLib.sh

This is the most basic library for authors. It defines a number of useful functions. The name of these functions is prefixed by `fw4ex_`.

fw4ex_transcode_carefully

This is a filter to transcode weird characters into XML and UTF-8 acceptable characters. Remember that every string (especially the strings coming from the student) going to appear in the grading report must pass through this filter so the generated XML is valid.

Several options are possible: -l numbers the lines. --g=N limits the output to N characters: if the output overflows, a series of three characters is emitted. --b=N limits the number of non-UTF8 bytes (the error code is 1). --s=N cuts lines every N characters.

The FW4EX_TRANSCODE_CAREFULLY_FLAGS variable may be set to add permanently options to fw4ex_transcode_carefully.

```
fw4ex_transcode_carefully () {
    expand | $FW4EX_BIN_DIR/transcode $FW4EX_TRANSCODE_CAREFULLY_FLAGS "$@"
}
```

For instance, if you want to show to the student the command he submitted, use the following snippet:

```
    echo $OPTIONS | fw4ex_transcode_carefully
```

fw4ex_win

This function emits a partial mark. These partial marks will be summed to form the final mark. A mark is a unique argument, a string, that may contain an arithmetic expression. Only two decimals will be retained for the partial mark to emit.

```
fw4ex_win() {
    local POINT=$( $FW4EX_LIB_DIR/win.pl "$@" )
    fw4ex_generate_fw4ex_element formula: "$@"
    cat <<EOF
<mark key='$FW4EX_JOB_ID' value='$POINT'>$POINT</mark>
EOF
}
```

Here are some examples of wins:

```
fw4ex_win 1               # to win 1
fw4ex_win 1/3             # to win 0.33
fw4ex_win 3-\(5.6\*0.5\)  # to win 0.2
fw4ex_win "3 - (5.6*0.5)" # to win 0.2
```

fw4ex_confine

The fw4ex_confine function allows an author to run some student's code in a confined mode so the student's code cannot exceed some limits (duration, output size (stdout or stderr)). Normally, an author should never run student's code out of a confined mode!

Pay attention, that the author's scripts are themselves confined by the FW4EX platform so an author may only devote resources to student's scripts within his own limits.

Normally, the maxcpu, maxout and maxerr limitations are already set for the author's script. If you want to set more precise limits (or hide additional environement variables) to run the student's script, use the FW4EX_STUDENT_LIMITS variable, for instance:

```
FW4EX_STUDENT_LIMITS='--maxout=5k --cpu=1'
```

The exit code of the confined script will be stored in the TMPDIR/.lastExitCode file (initially filled with 211). The code 222 is returned if the confined program was killed because it exceeds its specified duration. The code 228 if the confined program was killed because it exceeds the number of characters to be output on its stdout or stderr streams. Otherwise the the exit value of the confined process if it terminates naturally. See the confine utility for more details.

Please consider using `fw4ex_run_student_command` or `fw4ex_run_teacher_command` instead from the `comparisonLib.sh` library.

FUTURE DEBUG retirer -v si present

```
fw4ex_confine() {
    #fw4ex_generate_xml_comment "FW4EX_STUDENT_LIMITS=$FW4EX_STUDENT_LIMITS"
    echo -n 211 > $TMPDIR/.lastExitCode
    $FW4EX_BIN_DIR/confine \
        --exit-file $TMPDIR/.lastExitCode \
        $FW4EX_STUDENT_LIMITS \
        --fw4ex-hide \
        -- "$@"
}
```

fw4ex_generate_xml_comment

The `fw4ex_generate_xml_comment` function is a deprecated internal technical function used to insert additional information within grading report not immediately seeable by students. Use it only for debug as it may leak useful information towards students.

The `fw4ex_generate_xml_comment` function generates a comment paying attention not to generate a sequence of two dashes (--) within the comment.

```
fw4ex_generate_xml_comment () {
    echo "<!--" $( echo "$@" | sed -r 's/-{2,}/-/g' | \
        fw4ex_transcode_carefully ) "-->"
}
```

fw4ex_generate_fw4ex_element

This is an internal technical function that emits an `FW4EX` element. These elements are used to insert additional information within grading report and to resynchronize XML generation in case of generation problems.

```
fw4ex_generate_fw4ex_element () {
    echo "<FW4EX what='" $( echo "$@" | fw4ex_transcode_carefully ) "'/>"
}
```

8.8.3 moreLib.sh

This library defines additional functions that may be useful for authors. This file defines some internal functions that are not documented.

fw4ex_ensure_final_newline

This function appends a final newline to a file if this new newline is missing. This is sometimes useful to sanitize student's data files.

```
fw4ex_ensure_final_newline () {
    local FILE="$1"
    local CHAR=$( tail --bytes=1 < "$FILE" | od -w1 -c -An )
    case "$CHAR" in
        *'\n'*)
            return 0
            ;;
        *)
```

```
            echo >>"$FILE"
            return 1
            ;;
    esac
}
```

fw4ex_check_existence

This function checks whether a file exists (ie whether the student submits this file), is readable and not empty. Aborts (with the content of the ABORT variable) the current script if the file does not exist. Another setting for ABORT is return.

Verbalizes the check while doing it.

```
fw4ex_check_existence() {
    local ABORT=${ABORT:-exit}
    local FILE="$1"

    if [[ ! -f "$FILE" ]]
    then
        case "$FW4EX_LANG" in
            fr)
                cat <<EOF
<error>Je ne trouve pas de fichier nommé <code>$FILE</code>! </error>
EOF
                ;;
            en|*)
                cat <<EOF
<error>I cannot find a file named <code>$FILE</code>! </error>
EOF
                ;;
        esac
        $ABORT 51
    fi

    if [[ ! -r "$FILE" ]]
    then
        chmod a+r "$FILE"
    fi

    if [[ ! -s "$FILE" ]]
    then
        case "$FW4EX_LANG" in
            fr)
                cat <<EOF
<error>Votre fichier nommé <code>$FILE</code> est vide! </error>
EOF
                ;;
            en|*)
                cat <<EOF
<error>Your file, named <code>$FILE</code>, is empty! </error>
EOF
                ;;
        esac
        $ABORT 52
    fi
}
```

fw4ex_show_directory

List the given directory (by default, the current directory).

```
fw4ex_show_directory () {
    local DIR="$1"
    if [[ -n "$DIR" ]]
    then
        case "$FW4EX_LANG" in
            fr)
                echo "<p> Voici le contenu du répertoire <code>$DIR</code>: <pre>"
                ;;
            en|*)
                echo "<p> Here is the content of the <code>$DIR</code>
directory: <pre>"
                ;;
        esac
        (cd $DIR/ && ls -RgG ) | fw4ex_transcode_carefully
    else
        case "$FW4EX_LANG" in
            fr)
                echo "<p> Voici le contenu de votre répertoire: <pre>"
                ;;
            en|*)
                echo "<p> Here is the content of your directory: <pre>"
                ;;
        esac
        ls -RgG | fw4ex_transcode_carefully
    fi
    echo "</pre></p>"
}
```

fw4ex_compare_strings

Compare two strings using Levenshthein distance that is, the number of insertions/deletion/substitution needed to change the first (student's) string into the second (author's) one. Return that number in order to compute a partial mark. Don't give too long strings or the computation may take time.

```
fw4ex_compare_strings() {
    local STUDENT="$1"
    local TEACHER="$2"
    if false
    then
        # Il y a un probleme la-dessous (test 103c ???)
        perl -e "use Text::Levenshtein qw(distance);
                 print distance('$STUDENT', '$TEACHER'); "
    else
        # Pure perl             FUTURE a confiner a 5 secondes max ???
        $FW4EX_LIB_DIR/levenshtein.pl "$STUDENT" "$TEACHER"
    fi
}
```

fw4ex_compare_lines

Compare two files counting the number of insertions, deletion or substitution needed to change the first (student's) file into the second (author's) one. Return that number in order to compute a partial mark. Don't give too much lines or the computation may take time.

Additionnally, you may add other options to `diff` in the third argument. If the third argument is missing, it defaults to the value of the global variable `DIFF_AUTHOR_FLAGS`.

```
fw4ex_compare_lines() {
    local STUDENT="$1"
    local TEACHER="$2"
    local DIFF_AUTHOR_FLAGS="${3:-$DIFF_AUTHOR_FLAGS}"
    local DIFF_FLAGS='--unchanged-line-format= --old-line-format=O --new-line-format=N'
    local RESULT=$( diff $DIFF_AUTHOR_FLAGS $DIFF_FLAGS $STUDENT $TEACHER )
    #echo "<!-- $RESULT -->"
    echo ${#RESULT}
}
```

fw4ex_compare_integers

Compare two integers (for example two exit values). Return the distance between them (probably not a very meaningful distance for exit values) in order to compute the partial mark.

```
fw4ex_compare_integers() {
    local STUDENT="$1"
    local TEACHER="$2"
    echo $(( $TEACHER - $STUDENT )) | sed -re 's/-//'
}
```

The <expectations> element from the `fw4ex.xml` file automatizes the verification that the expected files are indeed present. The generated code uses the following four functions. All these functions only emit a text, they should not use 'exit'! An exit will automatically be performed after reporting a miss.

fw4ex_report_present_file

Generate a FW4EX comment telling if a file is present otherwise generate nothing.

```
fw4ex_report_present_file() {
    local FILE="$1"
    fw4ex_generate_fw4ex_element $FILE is present
}
```

fw4ex_report_missing_file

Notify the user that an expected file is missing. Returns true only if the file is really missing.

```
fw4ex_report_missing_file() {
    local FILE="$1"
    if [[ -f $FILE ]]
    then
        false
```

```
    else
        case "$FW4EX_LANG" in
            fr)
                cat <<EOF
<error>Je ne trouve pas de fichier nommé <code>$FILE</code>! </error>
EOF
                ;;
            en|*)
                cat <<EOF
<error> I cannot find a file named <code>$FILE</code>! </error>
EOF
                ;;
        esac
        true
    fi
}
```

fw4ex_report_present_directory

Generate a FW4EX comment telling if a directory is present otherwise generate nothing.

```
fw4ex_report_present_directory() {
    local DIR="$1"
    fw4ex_generate_fw4ex_element $DIR is present
}
```

fw4ex_report_missing_directory

Notify the user that an expected directory is missing. Returns true only if this directory is really missing.

```
fw4ex_report_missing_directory() {
    local DIR="$1"
    if [[ -f $DIR ]]
    then
        false
    else
        case "$FW4EX_LANG" in
            fr)
                cat <<EOF
<error>Je ne trouve pas de répertoire nommé <code>$DIR</code>! </error>
EOF
                ;;
            en|*)
                cat <<EOF
<error> I cannot find a directory named <code>$DIR</code>! </error>
EOF
                ;;
        esac
        true
    fi
}
```

8.8.4 imgLib.sh

This library defines additional functions to insert images within the grading report.

fw4ex_insert_image

This function embeds an image (designated as a file) within the grading report. The first argument is the file containing the image (png, jpeg images are well handled), the second argument is the text serving as ALT attribute. Attention, there are certain restriction on the size of the image in some browsers.

8.8.5 makefileLib.sh

This library is used when dealing with make or gcc. It defines a number of useful functions. The name of these functions is prefixed by `fw4ex_`.

fw4ex_is_relocatable

This predicate takes a filename as argument and checks whether it is a relocatable file (that is a `.o` file) or not. The check is made with the `file` utility (pay attention to the current LANG value).

```
fw4ex_is_relocatable () {
    local FILE="$1"
    case "$( file $FILE )" in
        *relocatable*)
            true
            ;;
        *)
            false
            ;;
    esac
}
```

fw4ex_is_library

This predicate takes a filename as argument and checks whether it is a library file (that is, a `.a` file). The check is made with the `file` utility (pay attention to the current LANG value).

```
fw4ex_is_library () {
    local FILE="$1"
    case "$( file $FILE )" in
        *archive*)
            true
            ;;
        *)
            false
            ;;
    esac
}
```

fw4ex_is_symbol_defined

This predicate checks whether a file (that may be given as argument to the `nm` utility) defines a symbol.

```
fw4ex_check_is_symbol_defined () {
    local FILE="$1"
    local NAME="$2"
    cat <<EOF
<p>Je cherche si <code>$FILE</code> définit bien
le nom <code>$NAME</code>.
EOF
    if nm "$FILE" | fgrep -q "T $NAME"
    then
        echo "C'est bien le cas!"
    else
        nm $file
        cat <<EOF
<error>Le symbole <code>$NAME</code> n'est pas défini dans le
fichier <code>$FILE</code>!</error>
EOF
    fi
    echo "</p>"
}
```

fw4ex_has_type

This predicate takes a filename and a string (the expected type of the file) and checks whether this file has this type. The type is determined with the `file` utility. For example, to check whether a file is a TeX DVI file, one should write

```
fw4ex_has_type someFile 'TeX DVI'
```

Pay attention, the `file` utility depends on the current LANG.

```
fw4ex_has_type () {
    local FILE=$1
    local TYPE="$2"
    if [[ ! -f $FILE ]]
    then
        cat <<EOF
<error>
Le fichier <code>$FILE</code> n'existe pas!
</error>
EOF
        false
    else
        #local MAGIC_NUMBER=$( od -Anone -c -N2 $FILE | tr -d ' ' )
        #if [[ "$MAGIC_NUMBER" = '367002' ]]
        file $FILE > $TMPDIR/file
        if grep -qi "$TYPE" < $TMPDIR/file
        then
            true
        else
            cat <<EOF
```

```
<error>
Votre fichier <code>$FILE</code> n'est pas un fichier de
type <code>$TYPE</code>, l'utilitaire <code>file</code> dit que c'est
<code>$( fw4ex_transcode_carefully < $TMPDIR/file )</code>.
</error>
EOF
            false
        fi
    fi
}
```

fw4ex_creation_hour

This small and internal function returns the creation time of a file but only the hour, minute and second part that is, something like `16:10:53.000000000`. This is useful to check whether `make` recreates a file or not.

```
fw4ex_creation_hour () {
    local FILE="$1"
    ls --full-time "$FILE" | awk '{print $7}'
}
```

fw4ex_is_impacted

This internal predicate takes two filenames: a target and a requisite and checks whether the target is rebuilt when the requisite is touched. This proves that the target is indeed dependent on the requisite.

```
fw4ex_is_impacted () {
    local TARGET="$1"
    local REQUISITE="$2"
    local OLDHOUR="$( fw4ex_creation_hour "$TARGET" )"
    sleep 1  # less than 1 second may be ignored
    touch "$REQUISITE"
    fw4ex_run_student_command $MAKE "$TARGET"
    local NEWHOUR="$( fw4ex_creation_hour "$TARGET" )"
    [ "$OLDHOUR" != "$NEWHOUR" ]
}
```

fw4ex_check_impact

This function takes, as arguments, a filename (a target) followed by a number of requisites. It checks and verbalizes (in French) whether the target depends on the requisite with help of the `fw4ex_is_impacted` predicate.

```
fw4ex_check_impact () {
    local TARGET="$1"
    shift
    local RESULT=true
    for REQUISITE in "$@"
    do
      cat <<EOF
<p>
Je vérifie si <code>$TARGET</code> dépend de
<code>$REQUISITE</code>. Je "touche" (comme l'on dit)
```

```
<code>$REQUISITE</code> puis je demande à reconstruire
<code>$MAKE $TARGET</code>.
EOF
      fw4ex_is_impacted "$TARGET" "$REQUISITE"
      local STATUS=$?
      fw4ex_show_student_raw_result
      if [[ $STATUS -eq 0 ]]
      then
          echo "Le fichier <code>$TARGET</code> a été reconstruit. "
      else
          cat <<EOF
Le fichier <code>$TARGET</code> n'a pas changé!
Il ne dépend donc pas de <code>$REQUISITE</code>.
EOF
          RESULT=false
      fi
    done
    echo "</p>"
    $RESULT
}
```

fw4ex_clean_target

Before testing that make builds a file, it is safer to be sure that the intended file does not exist. This function removes all the mentioned targets and verbalizes this fact.

```
fw4ex_clean_target () {
    for FILE in "$@"
    do
      if [[ -f "$FILE" ]]
      then
          cat <<EOF
<p>Je vois que vous avez un fichier <code>$FILE</code>,
je l'efface.</p>
EOF
          rm -f "$FILE"
      fi
    done
}
```

fw4ex_make_once

This function takes a target and runs the make command to build or rebuild that target. If FW4EX_SHOW_ERROR_CODE is true, the error code of the make is displayed if erroneous. The make command is run with fw4ex_run_student_command and as such leaves the stdout, stderr and error code as usual.

```
FW4EX_SHOW_ERROR_CODE=false

fw4ex_make_once () {
    cat <<EOF
<p>Avant de demander l'exécution de <code>$MAKE $@</code>, je demande
à voir ce qui va être fait avec <code>$MAKE -n --no-print-directory $@</code>:
<pre>
EOF
```

```
    $MAKE -n --no-print-directory "$@" | fw4ex_transcode_carefully
    cat <<EOF
</pre>
J'exécute maintenant la commande <code>$MAKE $@</code>.
EOF
    fw4ex_run_student_command $MAKE "$@"
    local errcode=$( cat $TMPDIR/.lastExitCode )
    if ${FW4EX_SHOW_ERROR_CODE:-false}
    then
        if [[ $errcode -ne 0 ]]
        then
            cat <<EOF
<warning>Votre commande <code>$MAKE $@</code> a retourné le code
$errcode.</warning>
EOF
            fw4ex_verbalize_error_code $TMPDIR/.lastExitCode
        fi
    fi
    echo "</p>"
    return $errcode
}
```

fw4ex_re_make

This predicate takes a target and rebuilds it with make. It returns true iff nothing is done that is, the target is up to date. This predicate verbalizes its actions in French.

```
fw4ex_re_make () {
    local TARGET="$1"
    shift
    local OLDHOUR="$( fw4ex_creation_hour $TARGET )"
    cat <<EOF
<p> Je demande à nouveau à voir ce qui sera fait avec la commande
<code>$MAKE -n --no-print-directory $TARGET $@</code>: <pre>
EOF
    $MAKE -n --no-print-directory $TARGET "$@" | \
        tee $TMPDIR/result.s | fw4ex_transcode_carefully
    cat <<EOF
</pre></p>
EOF
    # If the target is already up to date, don't run make!
    if grep -Eq "^make:.*$TARGET.*(is up to date|est .* jour)" < $TMPDIR/result.s
    then :
    else
        cat <<EOF
<p>J'exécute maintenant la commande <code>$MAKE $TARGET $@</code>.</p>
EOF
        fw4ex_run_student_command $MAKE $TARGET "$@"
        fw4ex_show_student_raw_result
        if ${FW4EX_SHOW_ERROR_CODE:-false}
        then
            local errcode=$( cat $TMPDIR/.lastExitCode )
            if [[ $errcode -ne 0 ]]
            then
```

```
                cat <<EOF
<warning>Votre commande <code>$MAKE $TARGET $@</code> a retourné le code
$errcode.</warning>
EOF
            fi
        fi
    fi

    if grep -Eq "^make:.*$TARGET.*(is up to date|est .* jour)" < $TMPDIR/result.s
    then
        cat <<EOF
<p>Le fichier <code>$TARGET</code> est bien à jour et rien n'a été
effectué: c'est parfait.</p>
EOF
        true
    else
        local NEWHOUR="$( fw4ex_creation_hour $TARGET )"
        if [[ "$OLDHOUR" == "$NEWHOUR" ]]
        then
            cat <<EOF
<p> L'utilitaire <code>$MAKE</code> a refait quelque-chose mais la date
de création de la cible <code>$TARGET</code> est toujours la même. </p>
EOF
            true
        else
            cat <<EOF
<error>Le fichier <code>$TARGET</code> a changé d'heure de
création. C'était $OLDHOUR, c'est maintenant $NEWHOUR.</error>
EOF
            false
        fi
    fi
}
```

fw4ex_show_current_directory

Display the content of the current directory. This is often useful before running a make command. This command also stores the content of the directory in the `TMPDIR/ls.before` file.

```
fw4ex_show_current_directory () {
   echo "<p> Voici le contenu du répertoire courant: <pre>"
   ls -gG | fw4ex_transcode_carefully
   ls -1 > $TMPDIR/ls.before
   echo "</pre></p>"
}
```

fw4ex_show_current_directory_after

Display the content of the current directory. This is often useful after running a make command. This command also stores the content of the directory in the `TMPDIR/ls.after` file.

```
fw4ex_show_current_directory_after () {
   echo "<p> Voici le nouveau contenu du répertoire courant: <pre>"
```

```
    ls -gG | fw4ex_transcode_carefully
    ls -1 > $TMPDIR/ls.after
    echo "</pre></p>"
}
```

8.8.6 comparisonLib.sh

This library defines functions used in various patterns. It requires the `basicLib.sh` and `moreLib.sh` libraries. All these functions have names prefixed by `fw4ex_`, they emit French sentences in UTF-8. Most of these functions are very simple so their source is given without much explanations.

fw4ex_show_command_and_options_from_file

```
fw4ex_show_command_and_options_from_file () {
    local COMMAND="$1"
    local OPTIONS="$2"
    cat <<EOF
<p>
Voici donc la commande que vous avez choisie:
<pre>
$( echo "$COMMAND $(cat $OPTIONS)" | fw4ex_transcode_carefully )
</pre></p>
EOF
}
```

fw4ex_show_script

Display a script but don't display files that are not recognized as text files. Also generate a special FW4EX tag in order to build an index of shown files.

```
FW4EX_SHOW_SCRIPT_PREFIX_GOOD="Voici donc votre réponse:"
fw4ex_show_script () {
    local SCRIPT="$1"
    if [[ 0 -eq $( wc -c < "$SCRIPT" ) ]]
    then
        echo "<p><error>Votre fichier est vide!</error></p>"
    else
        case "$(file "$SCRIPT")" in
          *text*)
            fw4ex_generate_fw4ex_element "file:///$SCRIPT"
            cat <<EOF
<p>
$FW4EX_SHOW_SCRIPT_PREFIX_GOOD
<pre>
EOF
            fw4ex_transcode_carefully --l < "$SCRIPT"
            cat <<EOF
</pre></p>
EOF
            ;;
        *)
            cat <<EOF
<warning> Votre fichier
```

```
<code>$( echo $SCRIPT | fw4ex_transcode_carefully )</code>
ne semble pas être un texte,
l'utilitaire <code>file</code> dit en effet que c'est un
<code>$( file "$SCRIPT" | fw4ex_transcode_carefully )</code>:
je ne tente donc pas de vous le montrer. </warning>
EOF
            ;;
        esac
    fi
}
```

fw4ex_check_executability

```
fw4ex_check_executability () {
    local SCRIPT="$1"
    if [[ -x "$SCRIPT" ]]
    then
        return 0
    else
        cat <<EOF
<error> L'énoncé demandait un script exécutable ce que n'est pas votre
réponse. Je rectifie les droits au passage. </error>
EOF
        chmod a+x "$SCRIPT"
        return 1
    fi
}
```

fw4ex_check_options_prefixed_by_command

```
fw4ex_check_options_prefixed_by_command () {
    local COMMAND="$1"
    local OPTIONS="$2"
    local OPT=$( cat $OPTIONS )
    case "$OPT" in
        "$COMMAND "*)
            cat <<EOF
<warning> L'énoncé demandait les <strong>options</strong> à placer après la
commande <code>$COMMAND</code>, vous avez néanmoins fait précéder ces
options du nom de cette commande ce qui n'est pas conforme à la consigne!
Le reste sera probablement donc erroné. À l'avenir, lisez
soigneusement les énoncés!
</warning>
EOF
            ;;
    esac
}
```

fw4ex_show_data_file

This function displays the content of a data file. It does not show the name of the data file only its content. It receives as first argument, the rank of the data file among the data files.

```
fw4ex_show_data_file () {
    local FLAG=
    if ${FW4EX_NUMBER_LINE:-false}
    then FLAG=--l
    fi
    local I="$1"
    local DATAFILE="$2"
    fw4ex_generate_fw4ex_element 'Essai' $I $( date -u +'%Y-%m-%dT%H:%M:%SZ' )
    if [[ -z "${DATAFORMATTER}" ]]
    then
        echo "<p> Voici le contenu du fichier $I à traiter: <pre>"
        fw4ex_transcode_carefully $FLAG < $DATAFILE
        echo "</pre></p>"
    else
        cat <<EOF
<p> Voici le contenu du fichier $I à traiter (que j'affiche avec
<code>$DATAFORMATTER</code>): <pre>
EOF
        ${DATAFORMATTER:-cat} < $DATAFILE | fw4ex_transcode_carefully $FLAG
        echo "</pre></p>"
    fi
}
```

fw4ex_show_directory_content

This function displays the content of the current directory. It does not show the name of this directory only its content. It receives as first argument, the rank of the data file among the data files and, as second argument, the options to use to invoke `ls`.

```
fw4ex_show_directory_content () {
    local I="$1"
    local LS_FLAGS="${2:-gG}"
    fw4ex_generate_fw4ex_element Épreuve $I
    cat <<EOF
<p> Voici le contenu du répertoire $I en lequel opérer: <pre>
EOF
    ls $LS_FLAGS | fw4ex_transcode_carefully
    cat <<EOF
</pre></p>
EOF
}
```

fw4ex_show_student_command

This function echoes the command that will be run.

```
fw4ex_show_student_command () {
    local COMMAND="$1"
    shift
    if $FW4EX_SHOW_COMMAND
    then
        cat <<EOF
<p> Je vais donc exécuter la commande suivante: <pre>
$( echo $COMMAND $@ | fw4ex_transcode_carefully )
```

```
</pre></p>
EOF
    fi
    if $DEBUG
    then
        fw4ex_generate_xml_comment "$COMMAND" "$@"
    fi
}
```

fw4ex_run_student_command

This function takes the (student's) command to run as first argument. The student's command is confined, stdout and stderr are preserved in temporary files. By default, the PATH is prepended with the HOME directory of the student though you may override this after setting the STUDENT_ADDITIONAL_PATH.

```
fw4ex_run_student_command () {
    local COMMAND="$1"
    shift
    chmod a+x "$COMMAND" 2>/dev/null
    (
        ( cd $TMPDIR/ && rm -f result.[st] result.n[st] ) 2>/dev/null
        PATH=${STUDENT_ADDITIONAL_PATH:-$HOME}:$PATH
        eval "fw4ex_confine $COMMAND ""$@" \
            2>$TMPDIR/err.s >$TMPDIR/result.s
        cp -p $TMPDIR/.lastExitCode $TMPDIR/.lastExitCode.s
        fw4ex_generate_xml_comment exitCode $( cat $TMPDIR/.lastExitCode.s )
        fw4ex_verbalize_error_code $TMPDIR/.lastExitCode.s
    )
}
```

fw4ex_run_student_command_and_options_from_file

This function takes the (student's) command and options filename in order to build the whole command to run.

```
fw4ex_run_student_command_and_options_from_file () {
    local COMMAND="$1"
    local OPTIONS="$2"
    fw4ex_run_student_command "$COMMAND" "$( cat $OPTIONS )"
}
```

fw4ex_verbalize_error_code

Analyse the content of the file holding the exit code of the last command ran by fw4ex_run_student_command. if the exit code is 222 or 228, then this is probably the confiner that ends the command, so we verbalise it in French.

```
FW4EX_VERBALIZE_ALL_ERRONEOUS_CODE=false
fw4ex_verbalize_error_code () {
    local FW4EX_EXIT_FILE="${1:-$TMPDIR/.lastExitCode.s}"
    local code=$( cat $FW4EX_EXIT_FILE )
    if [ $( wc -l < $FW4EX_EXIT_FILE ) -ge 1 ]
    then
        if [ $code -eq 222 ]  # too much cpu, report to author
```

```
        then
            echo "<error> Votre programme a été interrompu car
il prenait trop de temps! </error>"
        elif [ $code -eq 228 ]  # too much output, report to author
        then
            echo "<error> Votre programme a été interrompu car
il produisait trop de caractères! </error>"
        elif [ $code -eq 211 ] # program not started
        then
            echo "<error> Je n'ai pas réussi à lancer votre programme:
bizarre! </error>"
        else
        # confiner internal error: ignore it!
            :
        fi
    elif $FW4EX_VERBALIZE_ALL_ERRONEOUS_CODE
    then if [ $code -ne 0 ]
         then echo "<warning> Le code de retour de votre programme est
<code>$code</code> !? </warning>"
         fi
    fi
}
```

fw4ex_verbalize_error

Analyse an exit code. if the exit code is 222 or 228, then this is probably the confiner that ends the command, so we verbalise it in French.

```
fw4ex_verbalize_error () {
    local code="$1"
    if [ $code -eq 222 ]  # too much cpu, report to author
    then
        echo "<error> Votre programme a été interrompu car
il prenait trop de temps! </error>"
    elif [ $code -eq 228 ]  # too much output, report to author
    then
        echo "<error> Votre programme a été interrompu car
il produisait trop de caractères! </error>"
    elif [ $code -eq 211 ] # program not started
    then
        echo "<error> Je n'ai pas réussi à lancer votre programme:
bizarre! </error>"
    else
        # confiner internal error: ignore it!
        :
    fi
    if $FW4EX_VERBALIZE_ALL_ERRONEOUS_CODE
    then if [ $code -ne 0 ]
         then echo "<warning> Le code de retour de votre programme est
<code>$code</code> !? </warning>"
         fi
    fi
}
```

fw4ex_show_student_raw_result

Display the content of the stdout accumulated in the `$TMPDIR/result.s` file by the `fw4ex_run_student_command` script.

Sometimes, the result file is so large that it is inconvenient to display it without some additional formating (a presentation with several columns or a graphical presentation may be more suited). You may specify the formatter (by default `cat`) with the `FORMATTER` variable.

```
FW4EX_NUMBER_LINE=false

fw4ex_show_student_raw_result () {
    local FLAG=
    if ${FW4EX_NUMBER_LINE:-false}
    then FLAG=--l
    fi

    if [[ -z "${FORMATTER}" ]]
    then
        if [[ -s $TMPDIR/result.s ]]
        then (
                trap 'echo "</pre></p>"' 0
                echo "<p> Votre commande produit:<pre>"
                ${FORMATTER:-cat} < $TMPDIR/result.s | \
                    fw4ex_transcode_carefully $FLAG
             )
        else
            echo "<p>Votre commande n'a rien émis sur son flux de sortie.</p>"
        fi
    else (
            trap 'echo "</pre></p>"' 0
            cat <<EOF
<p> Votre commande produit un résultat (que j'affiche avec
<code>$FORMATTER</code>):<pre>
EOF
            ${FORMATTER:-cat} < $TMPDIR/result.s | \
                fw4ex_transcode_carefully $FLAG
          )
    fi
    fw4ex_show_student_stderr
}
```

fw4ex_show_student_stderr

Display the content of the stderr if not empty. The stderr was accumulated in the `$TMPDIR/err.s` file by the `fw4ex_run_student_command` script.

```
fw4ex_show_student_stderr () {
    if [[ -s $TMPDIR/err.s ]]
    then (

            trap 'echo "</pre></error>"' 0
            cat <<EOF
<error> Attention! Votre commande a aussi produit les anomalies suivantes
sur son flux d'erreur: <pre>
```

```
EOF
            fw4ex_transcode_carefully < $TMPDIR/err.s
         )
    fi
}
```

fw4ex_run_teacher_command

This command takes a command as first argument followed by additional arguments. It runs the command in a specialized PATH that is, the value of TEACHER_ADDITIONAL_PATH is prepended. The default value of TEACHER_ADDITIONAL_PATH is FW4EX_EXERCISE_DIR.

Similarly to fw4ex_run_student_command, the result is stored into $TMPDIR/result.t and the exit code is stored into $TMPDIR/.lastExitCode.t.

```
fw4ex_run_teacher_command () {
    local COMMAND="$1"
    shift
    case "$COMMAND" in
        */*)
            if [ ! -x "$COMMAND" ]
            then
                echo "Je ne vois pas de fichier $COMMAND executable!" 1>&2
            fi
            ;;
        *)
            true
            ;;
    esac
    if $DEBUG
    then
        fw4ex_generate_xml_comment "$COMMAND ""$@"
    fi
    (
        PATH=${TEACHER_ADDITIONAL_PATH:-$FW4EX_EXERCISE_DIR}:$PATH
        eval "fw4ex_confine $COMMAND ""$@" >$TMPDIR/result.t
        cp -p $TMPDIR/.lastExitCode $TMPDIR/.lastExitCode.t
        fw4ex_generate_xml_comment exitCode $( cat $TMPDIR/.lastExitCode )
    )
}
```

fw4ex_run_teacher_command_and_options_from_file

```
fw4ex_run_teacher_command_and_options_from_file () {
    local COMMAND="$1"
    local SOLUTION="$2"
    fw4ex_run_teacher_command "$COMMAND" "$( cat $SOLUTION )"
}
```

fw4ex_show_teacher_raw_result

Sometimes, the result file is so large that it is inconvenient to display it without some additional formating (a presentation with several columns or a graphical presentation may be more suited). You may specify the formatter (by default cat) with the FORMATTER variable.

```
fw4ex_show_teacher_raw_result () {
    local FLAG=
    if ${FW4EX_NUMBER_LINE:-false}
    then FLAG=--l
    fi
    (
        trap 'echo "</pre>"' 0
        cat <<EOF
<p> Ma commande produit:</p><pre>
EOF
        ${FORMATTER:-cat} < $TMPDIR/result.t | \
            fw4ex_transcode_carefully $FLAG
     )
}
```

fw4ex_normalize_and_show_results

This function normalizes the result files. If the NORMALIZER variable is missing, then cat (the identity function) is used otherwise the NORMALIZER is the name of the filter to normalize the standard input into the standard output. The same filter normalizes the student's result and the author's result.

To normalize the results is often needed before comparing them. The normalization may sort, remove undesirable characters, etc.

However, sometimes, the results files are so large that it is inconvenient to display them without some additional formating (a presentation with several columns or a graphical presentation may be more suited). You may specify the formatter (by default cat) with the FORMATTER variable.

The COMPARISON_DISPOSITION variable tells how to display the normalized results. If the lines of these files are short, then the landscape mode where the results are displayed side by side may be preferred. If missing this variable defaults to vertical.

```
fw4ex_normalize_and_show_results () {
    if [ ! -s $TMPDIR/result.s ]
    then touch $TMPDIR/result.s
    fi
    ${NORMALIZER:-cat} < $TMPDIR/result.s > $TMPDIR/result.ns 2>$TMPDIR/normerr.txt
    if [ ! -s $TMPDIR/result.t ]
    then touch $TMPDIR/result.t
    fi
    ${NORMALIZER:-cat} < $TMPDIR/result.t > $TMPDIR/result.nt

    if [[ -z "${NORMALIZER}" ]]
    then
        return
    fi

    local FLAG=
    if ${FW4EX_NUMBER_LINE:-false}
    then FLAG=--l
    fi

    cat <<EOF
<p> Je normalise votre résultat et le mien afin de les comparer
dans le respect de l'énoncé. </p>
EOF
```

```
    local MODE="${1:-${COMPARISON_DISPOSITION:-vertical}}"
    case "$MODE" in
        ################################
        landscape|horizontal|stacked)
            # stacked means tabs!
            cat <<EOF
<comparison><student><pre>
EOF
            ${FORMATTER:-cat} < $TMPDIR/result.ns | \
                fw4ex_transcode_carefully $FLAG
            echo "</pre>"
            if [[ -s $TMPDIR/normerr.txt ]]
            then
                cat <<EOF
<error> Attention! La normalisation de votre résultat a aussi produit
les anomalies suivantes sur son flux d'erreur: <pre>
EOF
                fw4ex_transcode_carefully < $TMPDIR/normerr.txt
                cat <<EOF
</pre></error>
EOF
            fi
            cat <<EOF
</student><teacher><pre>
EOF
            ${FORMATTER:-cat} < $TMPDIR/result.nt | \
                fw4ex_transcode_carefully $FLAG
            cat <<EOF
</pre></teacher></comparison>
EOF
            ;;
        ################################
        portrait|vertical|*)
            cat <<EOF
<p>Votre commande produit donc: <pre>
EOF
            ${FORMATTER:-cat} < $TMPDIR/result.ns | \
                fw4ex_transcode_carefully $FLAG
            echo "</pre></p>"
            if [[ -s $TMPDIR/normerr.txt ]]
            then
                cat <<EOF
<error> Attention! La normalisation de votre résultat a aussi produit
les anomalies suivantes sur son flux d'erreur: <pre>
EOF
                fw4ex_transcode_carefully < $TMPDIR/normerr.txt
                cat <<EOF
</pre></error>
EOF
            fi
            cat <<EOF
<p> Tandis que ma commande produit: <pre>
EOF
            ${FORMATTER:-cat} < $TMPDIR/result.nt | \
```

```
                fw4ex_transcode_carefully $FLAG
            cat <<EOF
</pre></p>
EOF
            ;;
    esac
}
```

fw4ex_compare_results

This is the generalized comparator. The `COMPARISON` variable tells which kind of comparison to perform. Possible values for `COMPARISON` are `string`, `line`, `code` or `int` (synonyms and plural forms are also possible). See the specialized comparators for further details.

```
fw4ex_compare_results () {
    case "${COMPARISON}" in
        char|character|characters|string|strings)
            fw4ex_compare_results_as_strings
            ;;
        line|lines)
            fw4ex_compare_results_as_lines
            ;;
        code|codes)
            fw4ex_compare_results_as_codes
            ;;
        int|integer|integers)
            fw4ex_compare_results_as_integers
            ;;
        *)
            echo "Cannot find such a comparison ($COMPARISON) ?" 1>&2
            $ABORT 101
            ;;
    esac
}
```

fw4ex_compare_results_as_strings

Compare the content of two files (a student file and a teacher file) and return the Levenshthein distance.

```
fw4ex_compare_results_as_strings () {
    local S_RESULT="$TMPDIR/result.ns"
    local T_RESULT="$TMPDIR/result.nt"
    if [[ ! -f $S_RESULT ]]
    then S_RESULT="$TMPDIR/result.s"
    fi
    if [[ ! -f $T_RESULT ]]
    then T_RESULT="$TMPDIR/result.t"
    fi
    cat <<EOF
<p> Je compare les deux résultats.
EOF
    local DISTANCE=$( fw4ex_compare_strings \
        "$(cat $S_RESULT)" \
```

```
        "$(cat $T_RESULT)" )
    fw4ex_verbalize_strings_comparison $DISTANCE
    echo "Vous gagnez "
    eval "fw4ex_win $WIN_FORMULA -- $DISTANCE"
    echo "point(s).</p>"
}

fw4ex_verbalize_strings_comparison () {
    local DISTANCE="$1"
    echo "$DISTANCE" > $TMPDIR/.distance
    if [[ "$DISTANCE" -eq 0 ]]
    then
        cat <<EOF
Bravo! Je ne vois pas de différence.
EOF
    elif [[ "$DISTANCE" -eq 1 ]]
    then cat <<EOF
Je trouve qu'il faut insérer/modifier/supprimer $DISTANCE caractère
pour passer de l'un à l'autre.
EOF
    else cat <<EOF
Je trouve qu'il faut insérer/modifier/supprimer $DISTANCE caractères
pour passer de l'un à l'autre.
EOF
    fi
}
```

fw4ex_compare_results_as_lines

```
fw4ex_compare_results_as_lines () {
    local S_RESULT="$TMPDIR/result.ns"
    local T_RESULT="$TMPDIR/result.nt"
    if [[ ! -f $S_RESULT ]]
    then S_RESULT="$TMPDIR/result.s"
    fi
    if [[ ! -f $T_RESULT ]]
    then T_RESULT="$TMPDIR/result.t"
    fi
    cat <<EOF
<p> Je compare les deux résultats.
EOF
    local DISTANCE=$( fw4ex_compare_lines "$S_RESULT" "$T_RESULT" )
    fw4ex_verbalize_lines_comparison $DISTANCE
    echo "Vous gagnez "
    eval "fw4ex_win $WIN_FORMULA -- $DISTANCE"
    echo "point(s).</p>"
}

fw4ex_verbalize_lines_comparison () {
    local DISTANCE="$1"
    echo "$DISTANCE" > $TMPDIR/.distance
    if [[ "$DISTANCE" -eq 0 ]]
    then
        cat <<EOF
```

```
Bravo! Je ne vois aucune différence.
EOF
    elif [[ "$DISTANCE" -eq 1 ]]
    then cat <<EOF
Je trouve qu'il faut insérer/modifier/supprimer $DISTANCE ligne
pour passer de l'un à l'autre.
EOF
    else cat <<EOF
Je trouve qu'il faut insérer/modifier/supprimer $DISTANCE lignes
pour passer de l'un à l'autre.
EOF
    fi
}
```

fw4ex_compare_results_as_codes

Compare two exit values. If the SUCCESS_FAILURE variable is true then only success (exit codes both 0) or failure (exit codes both different from 0) is checked.

```
fw4ex_compare_results_as_codes () {
    local SUCCESS_FAILURE=${SUCCESS_FAILURE:-false}
    local S_RESULT=$( cat $TMPDIR/.lastExitCode.s )
    local T_RESULT=$( cat $TMPDIR/.lastExitCode.t )
    cat <<EOF
<p> Je compare les deux codes de retour:
votre code de retour est $S_RESULT,
mon code de retour est $T_RESULT.
EOF
    if $SUCCESS_FAILURE
    then
        if [[ $S_RESULT -ne 0 ]]
        then
            echo "Votre commande s'est donc mal terminée."
            S_RESULT=1
        fi
        if [[ $T_RESULT -ne 0 ]]
        then
            echo "Ma commande s'est donc mal terminée."
            T_RESULT=1
        fi
    fi
    local DISTANCE=$( fw4ex_compare_integers $S_RESULT $T_RESULT )
    echo "Vous gagnez "
    eval "fw4ex_win $WIN_FORMULA -- $DISTANCE"
    echo "point(s).</p>"
}
```

fw4ex_compare_results_as_integers

This function compares the two normalized results files $TMPDIR/result.ns (normalized student's result) and $TMPDIR/result.nt (normalized teacher's result). These two files (or the non-normalized files $TMPDIR/result.s and $TMPDIR/result.t if they do not exist) are then filtered in order to remove any non numerical characters. The two resulting

numbers are then compared and their distance (the absolute value of their difference) is computed and given to the `WIN_FORMULA`.

```
fw4ex_compare_results_as_integers () {
    local S_RESULT="$TMPDIR/result.ns"
    local T_RESULT="$TMPDIR/result.nt"
    if [[ ! -f $S_RESULT ]]
    then S_RESULT="$TMPDIR/result.s"
    fi
    if [[ ! -f $T_RESULT ]]
    then T_RESULT="$TMPDIR/result.t"
    fi
    if $DEBUG
    then
        fw4ex_generate_xml_comment "S_RESULT=$S_RESULT"
        fw4ex_generate_xml_comment "T_RESULT=$T_RESULT"
    fi
    S_RESULT=$( tr -cd 0-9 < $S_RESULT | sed -re 's/^0*(.+)$/\1/' )
    T_RESULT=$( tr -cd 0-9 < $T_RESULT | sed -re 's/^0*(.+)$/\1/' )
    cat <<EOF
<p> Je compare les deux valeurs:
la vôtre est $S_RESULT,
la mienne est $T_RESULT.
EOF
    local DISTANCE=$( fw4ex_compare_integers $S_RESULT $T_RESULT )
    echo "Vous gagnez "
    eval "fw4ex_win $WIN_FORMULA -- $DISTANCE"
    echo "point(s).</p>"
}
```

8.9 Extra Libraries

This section lists some libraries that are not part of CodeGradX but might be of some interest for authors as it shows, after writing a few exercises, common functionalities that can be factorized in a library.

8.9.1 libILP.sh

Cette bibliothèque a été écrite pour une série d'exercices utilisant Java. Elle est ici commentée en français.

Cette bibliothèque factorise un certain nombre de fonctions utiles pour l'écriture d'exercices liés à ILP: un cours de compilation donné à l'UPMC.

Les classes compilées (celles de l'étudiant et les classes de test qui en dépendent) seront toutes stockées dans BINDIR

```
BINDIR=$TMPDIR/bin
mkdir -p $BINDIR
```

Les fichiers `.jar` communs avec Eclipse sont à placer dans le répertoire `Java/jars` de l'exercice similairement à l'organisation dans Eclipse:

```
JARDIR=$FW4EX_EXERCISE_DIR/Java/jars
```

Les classes de test en Java sont à placer, selon la convention adoptée pour ILP, en `Java/src/fr/upmc/ilp/ilpXtmeYTest/ProcessTest.java` Comme elles dépendent du code que doivent produire les étudiants, il faut les compiler. Il se peut qu'il y ait aussi besoin de fichiers dans `C/`, `Grammars/` Les programmes de tests sont en `Grammars/Samples/*.{xml,result,print}`

NOTA: `jing` attend que les grammaires incluses soient à côté de la grammaire incluante.

La plupart des exercices vont nécessiter les `.jar` suivants:

```
XMLUNITJAR=${JARDIR}/xmlunit-1.3.jar
JINGJAR=${JARDIR}/jing.jar
JCOMMANDERJAR=${JARDIR}/jcommander-1.18.jar
```

Il faut faire très attention au classpath afin que les étudiants ne masquent pas les classes correctes servant au test.

```
CP=$FW4EX_JUNIT_JAR:$JINGJAR:$JCOMMANDERJAR:$XMLUNITJAR
```

Ces deux variables sont des options supplémentaires pour `javac` et `java`:

```
JAVACFLAGS='-verbose'
JAVAFLAGS='-verbose'
```

fw4ex_ilp_compile_student_classes

Cette fonction compile (et verbalise) le fichier source java de l'étudiant, donné en argument. Généralement, l'argument ressemble à `fr/upmc/ilp/ilp2tme4/Process.java`. Si la compilation est correcte alors l'étudiant gagne `MARK4STUDENTCLASSCOMPILATION` points. Les resultats de compilation apparaitront en BINDIR.

Il peut être utile de positionner avant de compiler:

```
JAVACFLAGS='-sourcepath $HOME/Java/src/ '
```

```
MARK4STUDENTCLASSCOMPILATION=1
fw4ex_ilp_compile_student_classes () {
    local CLASSFILENAME="$@"
    cat <<EOF
<p> Je compile vos classes avec la commande: <pre>
$( echo javac $JAVACFLAGS -d $BINDIR -cp $CP:. $@ | \
     fw4ex_transcode_carefully --s=90 )
</pre></p>
EOF
    fw4ex_confine javac $JAVACFLAGS -Xmaxerrs 10 -Xmaxwarns 10 \
        -d $BINDIR \
        -cp $CP:. \
        $@ \
        >$TMPDIR/result.s 2>$TMPDIR/err.s
    fw4ex_verbalize_error_code $TMPDIR/.lastExitCode
    # javac ne produit rien sur le stdout
    fw4ex_show_student_stderr

    local code=$( cat $TMPDIR/.lastExitCode )
    if [ "$code" -gt 0 ]
    then
        fw4ex_generate_fw4ex_element "exit code $code"
        cat <<EOF
<p><error> La compilation a échoué. </error>
```

```
J'arrête tout! </p>
EOF
        exit
    else
        cat <<EOF
<p> La compilation a réussi, vous gagnez
$(fw4ex_win $MARK4STUDENTCLASSCOMPILATION) point. </p>
EOF
    fi
}
```

fw4ex_ilp_compile_test_classes

Cette fonction compile (et verbalise) le fichier source java de l'enseignant, donné en argument. Généralement, l'argument ressemble à quelque chose comme (c'est le test de l'enseignant): `$FW4EX_EXERCISE_DIR/Java/src/fr/upmc/ilp/ilp2tme4Test/ProcessTest.java` Si la compilation est correcte alors l'étudiant gagne `MARK4TESTCLASSCOMPILATION` points. Cette compilation est effectuée en utilisant prioritairement les classes de l'étudiant précédemment compilées en BINDIR.

Il peut être utile de positionner avant de compiler:

```
JAVACFLAGS='-sourcepath $FW4EX_EXERCISE_DIR/Java/src/ '
```

```
MARK4TESTCLASSCOMPILATION=20
fw4ex_ilp_compile_test_classes () {
    local CLASSFILENAME="$@"
    cat <<EOF
<p> Je compile maintenant mes classes de test avec la commande: <pre>
$( echo javac $JAVACFLAGS -d $BINDIR -cp $CP:$BINDIR $@ | \
     fw4ex_transcode_carefully --s=90 )
</pre></p>
EOF
    fw4ex_confine javac $JAVACFLAGS -Xmaxerrs 10 -Xmaxwarns 10 \
        -d $BINDIR \
        -cp $CP:$BINDIR \
        $@ \
        >$TMPDIR/result.s 2>$TMPDIR/err.s
    fw4ex_verbalize_error_code $TMPDIR/.lastExitCode
    # javac ne produit rien sur le stdout
    fw4ex_show_student_stderr

    local code=$( cat $TMPDIR/.lastExitCode )
    if [ "$code" -gt 0 ]
    then
        fw4ex_generate_fw4ex_element "exit code $code"
        cat <<EOF
<p><error> La compilation a échoué. </error>
J'arrête tout! </p>
EOF
        exit
    else
        cat <<EOF
<p> La compilation a réussi, vous gagnez encore
$(fw4ex_win $MARK4TESTCLASSCOMPILATION) point.
Voici donc le détail de toutes les classes compilées: </p>
```

```
EOF
        fw4ex_show_directory $BINDIR
    fi
}
```

fw4ex_ilp_compile_grammar

Cette fonction prend une grammaire .rnc et la compile en une grammaire .rng.

```
MARK4GRAMMARCOMPILATION=1
TRANG=$FW4EX_EXERCISE_DIR/Java/jars/trang.jar
fw4ex_ilp_compile_grammar () {
    local GRAMMAR="$1"
    cat <<EOF
<p> Je convertis votre grammaire .rnc en une grammaire .rng. </p>
EOF
    if [ -r ${GRAMMAR%.rnc}.rng ]
    then
        cat <<EOF
<p> Ah mais je vois que vous avez déjà <code>${GRAMMAR%.rnc}.rng</code>,
je la retire afin de vérifier votre .rnc. </p>
EOF
    fi

    case "$GRAMMAR" in
        */*)
            local GRAMMARDIR=${GRAMMAR%/*}
            local GRAMMARNAME=${GRAMMAR##*/}
            local HERE=`pwd`
            cd $GRAMMARDIR
            fw4ex_confine java -jar $TRANG \
                -i encoding=utf-8 -o encoding=utf-8 \
                $GRAMMARNAME ${GRAMMARNAME%.rnc}.rng \
                > $TMPDIR/result.s 2>$TMPDIR/err.s
            cd $HERE
            ;;
        *)
            fw4ex_confine java -jar $TRANG \
                -i encoding=utf-8 -o encoding=utf-8 \
                $GRAMMAR ${GRAMMAR%.rnc}.rng \
                > $TMPDIR/result.s 2>$TMPDIR/err.s
            ;;
    esac
    fw4ex_verbalize_error_code $TMPDIR/.lastExitCode
    fw4ex_show_student_stderr

    local code=$( cat $TMPDIR/.lastExitCode )
    if [ "$code" -gt 0 ]
    then
        fw4ex_generate_fw4ex_element "exit code $code"
        cat <<EOF
<p><error> La conversion a échoué. </error>
J'arrête tout! </p>
EOF
        exit
```

```
    else
        echo "<p> La conversion a réussi, voici vos grammaires: <pre>"
        case "$GRAMMAR" in
            */*)
                ls -lgG ${GRAMMAR%/*}/gramm* | \
                    fw4ex_transcode_carefully
                ;;
            *)
                ls -lgG gramm* | \
                    fw4ex_transcode_carefully
                ;;
        esac
        echo "</pre></p>"
        cat <<EOF
<p> Vous gagnez $(fw4ex_win $MARK4GRAMMARCOMPILATION) points. </p>
EOF
    fi
}
```

fw4ex_ilp_analyze_results

Cette fonction analyse et verbalise le résultat d'un test JUnit. Selon la façon dont le test est lancé, une dernière ligne indique le nombre de tests réussis ou ratés. Si l'on utilise `org.fw4ex.junit`, le nombre d'assertions réussies est aussi imprimé. Le nombre de points gagnés est `FACTOR` fois le nombre de tests réussis. `FACTOR` est l'argument de cette fonction (et par défaut vaut 1).

```
fw4ex_ilp_analyze_results () {
    local FACTOR="${1:-1}"

    fw4ex_ilp_extract_results
    fw4ex_ilp_verbalize_results "$FACTOR"
}

fw4ex_ilp_verbalize_results () {
    local FACTOR="${1:-1}"

    if (( "$FAILURES" == 0 ))
    then
        cat <<EOF
<p> L'exécution s'est très bien passé, vous gagnez
$(fw4ex_win "$FACTOR * $TESTS" ) point. </p>
EOF
    elif (( "$FAILURES" >= "$TESTS" ))
    then
        cat <<EOF
<error> Tous les tests ont échoué! </error>
<p> Vous ne gagnez aucun point. </p>
EOF
        exit
    else
        cat <<EOF
<error> Parmi les $TESTS tests, $FAILURES ont échoué! </error>
<p> Vous ne gagnez que $(fw4ex_win "$FACTOR * ( $TESTS - $FAILURES )")
points. </p>
```

```
EOF
        exit
    fi
}
```

8.10 Patterns

Since I do not come with a simple but sensible naming scheme for these patterns, I name them after French towns (there is an hidden pattern in choosing these names). Here follows a quick coded sketch of their main characteristics. They mainly differ in how the whole command is built, how the data file is fed (via stdin or argument), whether to jump to some directory, etc. In the following table, student's input appear in capitals.

Name	**Sketch**
BourgEnBresse	`$command $OPTIONS < $data`
Laon	`$COMMAND $command_flags < $data`
Moulin	`cd $datadir/ && $command $OPTIONS <$command_stdin`
Digne	`cd $datadir/ && $COMMAND <$command_stdin`
Gap	`$COMMAND $data <$command_stdin`
Nice	`$COMMAND $(cat $data) <$command_stdin`

8.10.1 BourgEnBresse.sh

This pattern corresponds to an exercise that asks for a set of options (a string) to tailor a command so it behaves as specified by the stem. For instance, what options to the Unix command `tr` may convert lower cases letters to upper case letters ? An answer is `a-z A-Z` but not `tr a-z A-Z` since the options are required but not the command (`tr`) which is already known.

This pattern gauges the student's answer by comparison to the author's solution. The command receives data files on its stdin, the result is in the stdout. To sum up and using the variables documented below, this pattern may be roughly characterized as:

```
$COMMAND $( cat $OPTIONS ) < $data
```

This pattern emits a grading report in French (encoded as UTF-8). Here is a short example:

```
COMMAND=tr
NORMALIZER=removeTrailingBlanks
COMPARISON=char
TOTAL_WIN=10
WIN_FORMULA='triangular 0 0 5 $TOTAL_WIN/$DATA_NUMBER'

removeTrailingBlanks () {
   sed -e 's/ *$//'
}

source $FW4EX_LIB_DIR/Patterns/BourgEnBresse.sh
```

Here are the parameters to configure this pattern.

COMMAND

This variable defines the program to run. The final command will be made of the program followed by the options proposed by the student.

PRECOMMAND

This variable if set tells how to transform the data to be fed to COMMAND. Examples of PRECOMMAND may be `sort` or `cut | sort`.

OPTIONS

This variable specifies the name of the file that contains the string of options (a single line). Pay attention, the variable does not hold the options but the filename where are stored the options. By default, this file is named `options`.

```
OPTIONS=${OPTIONS:-options}
```

SOLUTION

This variable specifies the name of the file that contains a correct solution that is a string of options achieving the desired behavior. By default, this pattern assumes that a `perfect` pseudo-job exists that contains the appropriate file named by the previous variable `OPTIONS`.

```
SOLUTION=${SOLUTION:-$FW4EX_EXERCISE_DIR/pseudos/perfect/$OPTIONS}
```

CHECK_OPTIONS

A common mistake is to prepend options with the name of the command. The stem should ask for options not for the whole command. See pattern \ref{pattern:Digne} for that. By default, this variable is `true` and signals the mistake. Set this variable to `false` if this is not desired.

Whether the mistake is signalled or not, the grading process continues.

```
CHECK_OPTIONS=${CHECK_OPTIONS:-true}
```

DATA_DIR

This variable is the name of the directory that contains the test files. By default, test files are located in the `tests/` directory of the tar gzipped exercise. This directory should be defined with an absolute pathname.

```
DATA_DIR=${DATA_DIR:-$FW4EX_EXERCISE_DIR/tests}
```

DATA_SUFFIX

This variable tells the suffix the test files have. By default, the suffix is `data`. Any file with that suffix in the `DATA_DIR` directory is a test file. Test files are used in alphabetic order.

The number of tests is therefore the number of files in `$DATA_DIR/*.$DATA_SUFFIX`. This number will be computed by the pattern and set as the value of the `DATA_NUMBER` variable.

```
DATA_SUFFIX=${DATA_SUFFIX:-data}
```

NORMALIZER

This variable contains the name of the command (program + options or internal bash function as shown in the synopsis above) that normalizes the output of the programs to compare. You are not required to define this variable if you don't need normalization.

COMPARISON

This variable contains a word that defines the kind of comparison.

The `char` comparison computes the Levenshtein distance between the student's answer and the author's answer. The Levenshtein distance should not be used to compare too long strings.

The `line` comparison uses the `diff` utility to count the number of dissimilar lines.

The `code` comparison compares the exit codes of the student's and author's commands.

The `int` comparison compares the stdout of the student's and author's commands. These stdout are assumed to be integers.

The `COMPARISON` variable is there to ease switching from one comparison method to another. If you're sure you may remove that intermediate and patch the script below.

```
COMPARISON=${COMPARISON:-char}
```

TOTAL_WIN

This variable defines the maximal mark that may be won. Given the nature of the pattern, any test case allows the student to win `TOTAL_WIN/DATA_NUMBER` (rounded to two decimals only). To avoid embarrassing rounding, avoid a `TOTAL_WIN` of 1 with only 3 tests. The total mark will amount to 0.99 instead of 1.

WIN_FORMULA

This variable defines how to compute the number of points a student wins for one test file. The comparison (determined by COMPARISON) computes a distance: 0 is perfection (student's and author's normalized answers are the same). The `WIN_FORMULA` variable defines how to convert the distance into a mark. It defines the first arguments of a call to `win.pl`. See documentation of this utility for more details.

FW4EX_SHOW_COMMAND

This boolean variable controls whether to display the command to run.

```
FW4EX_SHOW_COMMAND=${FW4EX_SHOW_COMMAND:-true}
```

Script

The script is sufficiently short to be shown. It uses the `basicLib.sh`, `moreLib.sh` and `comparisonLib.sh` libraries. The presence of the OPTIONS file is already checked.

```
 # Show the command that will be run to the student:

if [[ -n "$PRECOMMAND" ]]
then
    fw4ex_show_command_and_options_from_file "$PRECOMMAND | $COMMAND" "$OPTIONS"
else
    fw4ex_show_command_and_options_from_file "$COMMAND" "$OPTIONS"
fi

 # Check if students prefix their options by the name of the command:

if $CHECK_OPTIONS
then
    fw4ex_check_options_prefixed_by_command "$COMMAND" "$OPTIONS"
fi

 # For every data file, run student's and author's program and compare:

DATA_NUMBER=$( ls -1 $DATA_DIR/*.${DATA_SUFFIX} | wc -l )
if [[ $DATA_NUMBER -eq 0 ]] ; then exit ; fi
cat <<EOF
<p>
Je vais comparer votre solution et la mienne sur $DATA_NUMBER fichiers
de données.
</p>
EOF

I=0
 # Iterate on every data file
for data in $DATA_DIR/*.${DATA_SUFFIX}
do
    I=$(( $I+1 ))
    (
        trap 'echo "</section>"' 0
        # Describe the test ie show the content of the data file:
        echo "<section rank='$I'>"
        fw4ex_show_data_file $I "$data"

        # Run student's command:
        if [[ -n "$PRECOMMAND" ]]
        then
            # Show student's command:
            fw4ex_show_student_command \
                "cat $data | $PRECOMMAND | $COMMAND" "$(cat $OPTIONS)"
            cat "$data" | eval $PRECOMMAND | \
                fw4ex_run_student_command_and_options_from_file \
                   "$COMMAND" "$OPTIONS"
        else
            # Show student's command:
            fw4ex_show_student_command "$COMMAND" "$(cat $OPTIONS)" "< $data"
            fw4ex_run_student_command_and_options_from_file \
```

```
            "$COMMAND" "$OPTIONS" < "$data"
        fi
        # and show raw result of this command:
        fw4ex_show_student_raw_result
        # Possible hook for authors:
        fw4ex_after_student_run_hook $I "$data"

        # Run author's command. If this command is erroneous, errors
        # will be emitted on stderr and sent to the author.
        if [[ -n "$PRECOMMAND" ]]
        then
            cat "$data" | eval $PRECOMMAND | \
                fw4ex_run_teacher_command_and_options_from_file \
                   "$COMMAND" "$SOLUTION"
        else
            fw4ex_run_teacher_command_and_options_from_file \
                "$COMMAND" "$SOLUTION" < "$data"
        fi
        fw4ex_show_teacher_raw_result
        fw4ex_after_teacher_run_hook $I "$data"

        # Normalize then show raw results:
        if [[ -n "$NORMALIZER" ]]
        then
            fw4ex_normalize_and_show_results
        fi

        # Compare normalized results and determine the win:
        fw4ex_compare_results
    )
done
```

8.10.2 Laon.sh

This pattern corresponds to an exercise that asks for a command. For instance, what command may convert lower cases letters to upper case letters ?

This pattern gauges the student's answer by comparison to the author's solution. The command receives data files on its stdin, the result is in the stdout. To sum up and using the variables documented below, this pattern may be roughly characterized as:

```
$( cat $COMMAND ) $command_flags < $data
```

This pattern emits a grading report in French (encoded as UTF-8). Here is a short example:

```
NORMALIZER=removeTrailingBlanks
COMPARISON=line
TOTAL_WIN=10
WIN_FORMULA='triangular 0 0 5 $TOTAL_WIN/$DATA_NUMBER'

removeTrailingBlanks () {
   sed -e 's/ *$//'
}

source $FW4EX_LIB_DIR/Patterns/Laon.sh
```

Here are the parameters to configure this pattern.

COMMAND

This variable specifies the name of the file that contains the script to run. By default, this file is named `command`.

```
COMMAND=${COMMAND:-command}
```

COMMAND_FLAGS

This variable specifies the additional options to run `COMMAND`. By default, this variable is empty.

```
COMMAND_FLAGS=${COMMAND_FLAGS:-}
```

SOLUTION

This variable specifies the name of the file that contains a correct solution that is a command achieving the desired behavior. By default, this pattern assumes that a `perfect` pseudo-job exists that contains the appropriate file named by the previous variable `COMMAND`.

```
SOLUTION=${SOLUTION:-$FW4EX_EXERCISE_DIR/pseudos/perfect/$COMMAND}
```

DATA_DIR

This variable is the name of the directory that contains the test files. By default, test files are located in the `tests/` directory of the tar gzipped exercise. This directory should be defined with an absolute pathname.

```
DATA_DIR=${DATA_DIR:-$FW4EX_EXERCISE_DIR/tests}
```

DATA_SUFFIX

This variable tells the suffix the test files have. By default, the suffix is `data`. Any file with that suffix in the `DATA_DIR` directory is a test file. Test files are used in alphabetic order.

The number of tests is therefore the number of files in `$DATA_DIR/*.$DATA_SUFFIX`. This number will be computed by the pattern and set as the value of the `DATA_NUMBER` variable.

```
DATA_SUFFIX=${DATA_SUFFIX:-data}
```

NORMALIZER

This variable contains the name of the command (program + options or internal bash function as shown in the synopsis above) that normalizes the output of the programs to compare. You are not required to define this variable if you don't need normalization.

COMPARISON

This variable contains a word that defines the kind of comparison.

The `char` comparison computes the Levenshtein distance between the student's answer and the author's answer. The Levenshtein distance should not be used to compare too long strings.

The line comparison uses the diff utility to count the number of dissimilar lines.

The code comparison compares the exit codes of the student's and author's commands.

The int comparison compares the stdout of the student's and author's commands. These stdout are assumed to be integers.

The COMPARISON variable is there to ease switching from one comparison method to another. If you're sure you may remove that intermediate and patch the script below.

```
COMPARISON=${COMPARISON:-char}
```

TOTAL_WIN

This variable defines the maximal mark that may be won. Given the nature of the pattern, any test case allows the student to win TOTAL_WIN/DATA_NUMBER (rounded to two decimals only). To avoid embarrassing rounding, avoid a TOTAL_WIN of 1 with only 3 tests. The total mark will amount to 0.99 instead of 1.

WIN_FORMULA

This variable defines how to compute the number of points a student wins for one test file. The comparison (determined by COMPARISON) computes a distance: 0 is perfection (student's and author's normalized answers are the same). The WIN_FORMULA variable defines how to convert the distance into a mark. It defines the first arguments of a call to win.pl. See documentation of this utility for more details.

FW4EX_SHOW_SCRIPT

This boolean variable controls whether the student's script will be displayed or not. Byt default, the script is displayed.

```
FW4EX_SHOW_SCRIPT=${FW4EX_SHOW_SCRIPT:-true}
```

FW4EX_SHOW_COMMAND

This boolean variable controls whether to display the command to run.

```
FW4EX_SHOW_COMMAND=${FW4EX_SHOW_COMMAND:-true}
```

Script

The script is sufficiently short to be shown. It uses the basicLib.sh, moreLib.sh and comparisonLib.sh libraries. The presence of the COMMAND file is already checked.

```
 # Show the command that will be run to the student:

if $FW4EX_SHOW_SCRIPT
then
    fw4ex_show_script "$COMMAND"
fi
```

```
 # For every data file, run student's and author's program and compare:

DATA_NUMBER=$( ls -1 $DATA_DIR/*.${DATA_SUFFIX} | wc -l )
if [[ $DATA_NUMBER -eq 0 ]] ; then exit ; fi
cat <<EOF
<p>
Je vais comparer votre solution et la mienne sur $DATA_NUMBER fichiers
de données.
</p>
EOF

I=0
 # Iterate on every data file
for data in $DATA_DIR/*.${DATA_SUFFIX}
do
    I=$(( $I+1 ))
    (
        trap 'echo "</section>"' 0
        # Describe the test ie show the content of the data file:
        echo "<section rank='$I'>"
        fw4ex_show_data_file $I "$data"

        # Show student's command:
        fw4ex_show_student_command "$COMMAND" $COMMAND_FLAGS "< $data"
        # Run student's command:
        fw4ex_run_student_command "$COMMAND" $COMMAND_FLAGS < "$data"
        # and show raw result of this command:
        fw4ex_show_student_raw_result
        # Possible hook for authors:
        fw4ex_after_student_run_hook $I "$data"

        # Run author's command. If this command is erroneous, errors
        # will be emitted on stderr and sent to the author.
        fw4ex_run_teacher_command "$SOLUTION" $COMMAND_FLAGS < "$data"
        fw4ex_show_teacher_raw_result
        fw4ex_after_teacher_run_hook $I "$data"

        # Normalize then show raw results:
        if [[ -n "$NORMALIZER" ]]
        then
            fw4ex_normalize_and_show_results
        fi

        # Compare normalized results and determine the win:
        fw4ex_compare_results
    )
done
```

8.10.3 Moulins.sh

This pattern corresponds to an exercise that asks for a set of options (a string) to tailor a command so it behaves as specified by the stem. The command will be run in various directories.

This pattern gauges the student's answer by comparison to the author's solution. The command receives data files on its stdin, the result is in the stdout. To sum up and using the variables documented below, this pattern may be roughly characterized as:

```
        cd $datadir/ && $COMMAND $( cat $OPTIONS )
```

This pattern emits a grading report in French (encoded as UTF-8). Here is a short example:

```
COMMAND=tr
NORMALIZER=removeTrailingBlanks
COMPARISON=char
TOTAL_WIN=10
WIN_FORMULA='triangular 0 0 5 $TOTAL_WIN/$DATA_NUMBER'

removeTrailingBlanks () {
   sed -e 's/ *$//'
}

source $FW4EX_LIB_DIR/Patterns/Moulin.sh
```

Here are the parameters to configure this pattern.

COMMAND

This variable defines the program to run. The final command will be made of the program followed by the options proposed by the student.

OPTIONS

This variable specifies the name of the file that contains the string of options (a single line). Pay attention, the variable does not hold the options but the filename where are stored the options. By default, this file is named `options`.

```
OPTIONS=${OPTIONS:-options}
```

COMMAND_STDIN

This variable specifies the standard input to be given to `COMMAND`. By default, this is `/dev/null`.

```
COMMAND_STDIN=${COMMAND_STDIN:-/dev/null}
```

SOLUTION

This variable specifies the name of the file that contains a correct solution that is a string of options achieving the desired behavior. By default, this pattern assumes that a `perfect` pseudo-job exists that contains the appropriate file named by the previous variable `OPTIONS`.

```
SOLUTION=${SOLUTION:-$FW4EX_EXERCISE_DIR/pseudos/perfect/$OPTIONS}
```

CHECK_OPTIONS

A common mistake is to prepend options with the name of the command. The stem should ask for options not for the whole command. See pattern \ref{pattern:Digne} for that. By default, this variable is `true` and signals the mistake. Set this variable to `false` if this is not desired.

Whether the mistake is signalled or not, the grading process continues.

```
CHECK_OPTIONS=${CHECK_OPTIONS:-true}
```

DATA_DIR

This variable is the name of the directory that contains the test directories. By default, test directories are located in the `tests/` directory of the tar gzipped exercise. This directory should be defined with an absolute pathname.

```
DATA_DIR=${DATA_DIR:-$FW4EX_EXERCISE_DIR/tests}
```

DATA_SUFFIX

This variable tells the suffix the test files have. By default, the suffix is `d`. Any directory with that suffix in the `DATA_DIR` directory is a test directory. Test directories are used in alphabetic order.

The number of tests is therefore the number of directories in `$DATA_DIR/*.$DATA_SUFFIX/`. This number will be computed by the pattern and set as the value of the `DATA_NUMBER` variable.

```
DATA_SUFFIX=${DATA_SUFFIX:-d}
```

NORMALIZER

This variable contains the name of the command (program + options or internal bash function as shown in the synopsis above) that normalizes the output of the programs to compare. You are not required to define this variable if you don't need normalization.

COMPARISON

This variable contains a word that defines the kind of comparison.

The `char` comparison computes the Levenshtein distance between the student's answer and the author's answer. The Levenshtein distance should not be used to compare too long strings.

The `line` comparison uses the `diff` utility to count the number of dissimilar lines.

The `code` comparison compares the exit codes of the student's and author's commands.

The `int` comparison compares the stdout of the student's and author's commands. These stdout are assumed to be integers.

The `COMPARISON` variable is there to ease switching from one comparison method to another. If you're sure you may remove that intermediate and patch the script below.

```
COMPARISON=${COMPARISON:-char}
```

TOTAL_WIN

This variable defines the maximal mark that may be won. Given the nature of the pattern, any test case allows the student to win `TOTAL_WIN/DATA_NUMBER` (rounded to two decimals only). To avoid embarrassing rounding, avoid a `TOTAL_WIN` of 1 with only 3 tests. The total mark will amount to 0.99 instead of 1.

WIN_FORMULA

This variable defines how to compute the number of points a student wins for one test file. The comparison (determined by COMPARISON) computes a distance: 0 is perfection (student's and author's normalized answers are the same). The WIN_FORMULA variable defines how to convert the distance into a mark. It defines the first arguments of a call to win.pl. See documentation of this utility for more details.

FW4EX_SHOW_COMMAND

This boolean variable controls whether to display the command to run.

```
FW4EX_SHOW_COMMAND=${FW4EX_SHOW_COMMAND:-true}
```

Script

The script is sufficiently short to be shown. It uses the basicLib.sh, moreLib.sh and comparisonLib.sh libraries. The presence of the OPTIONS file is already checked.

```
 # Show the command that will be run to the student:

fw4ex_show_command_and_options_from_file "$COMMAND" "$OPTIONS"

 # Check if students prefix their options by the name of the command:

if $CHECK_OPTIONS
then
    fw4ex_check_options_prefixed_by_command "$COMMAND" "$OPTIONS"
fi

 # For every data file, run student's and author's program and compare:

DATA_NUMBER=$( ls -d1 $DATA_DIR/*.${DATA_SUFFIX}/ | wc -l )
if [[ $DATA_NUMBER -eq 0 ]] ; then exit ; fi
cat <<EOF
<p>
Je vais comparer votre solution et la mienne sur $DATA_NUMBER
répertoires de données.
</p>
EOF

I=0
 # Iterate on every data file
for datadir in $DATA_DIR/*.${DATA_SUFFIX}/
do
    I=$(( $I+1 ))
    (
        trap 'echo "</section>"' 0
        OPTIONS=$(pwd)/$OPTIONS
        cd $datadir
        # Describe the test ie show the content of the data directory:
        echo "<section rank='$I'>"
        fw4ex_show_directory_content $I "$datadir"
```

```
        # Show student's command:
        fw4ex_show_student_command "$COMMAND" "$( cat $OPTIONS )" \
            " < $COMMAND_STDIN"
        # Run student's command:
        fw4ex_run_student_command_and_options_from_file \
            "$COMMAND" "$OPTIONS" < $COMMAND_STDIN
        # and show raw result of this command:
        fw4ex_show_student_raw_result
        # Possible hook for authors:
        fw4ex_after_student_run_hook $I "$datadir"

        # Run author's command. If this command is erroneous, errors
        # will be emitted on stderr and sent to the author.
        fw4ex_run_teacher_command_and_options_from_file \
            "$COMMAND" "$SOLUTION" < $COMMAND_STDIN
        fw4ex_show_teacher_raw_result
        fw4ex_after_teacher_run_hook $I "$datadir"

        # Normalize then show raw results:
        if [[ -n "$NORMALIZER" ]]
        then
            fw4ex_normalize_and_show_results
        fi

        # Compare normalized results and determine the win:
        fw4ex_compare_results
    )
done
```

8.10.4 Digne.sh

This pattern corresponds to an exercise that asks for a command. The command will be run in various directories.

This pattern gauges the student's answer by comparison to the author's solution. The command is run within various directories, the result is in the stdout. To sum up and using the variables documented below, this pattern may be roughly characterized as:

```
cd $datadir/ && $( cat $COMMAND )
```

This pattern emits a grading report in French (encoded as UTF-8). Here is a short example:

```
NORMALIZER=removeTrailingBlanks
COMPARISON=char
TOTAL_WIN=10
WIN_FORMULA='triangular 0 0 5 $TOTAL_WIN/$DATA_NUMBER'

removeTrailingBlanks () {
   sed -e 's/ *$//'
}

source $FW4EX_LIB_DIR/Patterns/Digne.sh
```

Here are the parameters to configure this pattern.

COMMAND

This variable specifies the name of the file that contains the script to run. By default, this file is named `command`.

```
COMMAND=${COMMAND:-command}
```

COMMAND_STDIN

This variable specifies the standard input to be given to `COMMAND`. By default, this is `/dev/null`.

```
COMMAND_STDIN=${COMMAND_STDIN:-/dev/null}
```

COMMAND_FLAGS

This variable specifies the additional options to run `COMMAND`. By default, this variable is empty.

```
#cut
COMMAND_FLAGS=${COMMAND_FLAGS:-}
```

SOLUTION

This variable specifies the name of the file that contains a correct solution that is a script achieving the desired behavior. By default, this pattern assumes that a `perfect` pseudo-job exists that contains the appropriate file named by the previous variable `COMMAND`.

```
SOLUTION=${SOLUTION:-$FW4EX_EXERCISE_DIR/pseudos/perfect/$COMMAND}
```

DATA_DIR

This variable is the name of the directory that contains the test directories. By default, test directories are located in the `tests/` directory of the tar gzipped exercise. This directory should be defined with an absolute pathname.

```
DATA_DIR=${DATA_DIR:-$FW4EX_EXERCISE_DIR/tests}
```

DATA_SUFFIX

This variable tells the suffix the test files have. By default, the suffix is `d`. Any directory with that suffix in the `DATA_DIR` directory is a test directory. Test directories are used in alphabetic order.

The number of tests is therefore the number of directories in `$DATA_DIR/*.$DATA_SUFFIX/`. This number will be computed by the pattern and set as the value of the `DATA_NUMBER` variable.

```
DATA_SUFFIX=${DATA_SUFFIX:-d}
```

NORMALIZER

This variable contains the name of the command (program + options or internal bash function as shown in the synopsis above) that normalizes the output of the programs to compare. You are not required to define this variable if you don't need normalization.

COMPARISON

This variable contains a word that defines the kind of comparison.

The `char` comparison computes the Levenshtein distance between the student's answer and the author's answer. The Levenshtein distance should not be used to compare too long strings.

The `line` comparison uses the `diff` utility to count the number of dissimilar lines.

The `code` comparison compares the exit codes of the student's and author's commands.

The `int` comparison compares the stdout of the student's and author's commands. These stdout are assumed to be integers.

The `COMPARISON` variable is there to ease switching from one comparison method to another. If you're sure you may remove that intermediate and patch the script below.

```
COMPARISON=${COMPARISON:-char}
```

TOTAL_WIN

This variable defines the maximal mark that may be won. Given the nature of the pattern, any test case allows the student to win `TOTAL_WIN/DATA_NUMBER` (rounded to two decimals only). To avoid embarrassing rounding, avoid a `TOTAL_WIN` of 1 with only 3 tests. The total mark will amount to 0.99 instead of 1.

WIN_FORMULA

This variable defines how to compute the number of points a student wins for one test file. The comparison (determined by COMPARISON) computes a distance: 0 is perfection (student's and author's normalized answers are the same). The `WIN_FORMULA` variable defines how to convert the distance into a mark. It defines the first arguments of a call to `win.pl`. See documentation of this utility for more details.

FW4EX_SHOW_SCRIPT

This boolean variable controls whether the student's script will be displayed or not. By default, the script is displayed.

```
FW4EX_SHOW_SCRIPT=${FW4EX_SHOW_SCRIPT:-true}
```

FW4EX_SHOW_COMMAND

This boolean variable controls whether to display the command to run.

```
FW4EX_SHOW_COMMAND=${FW4EX_SHOW_COMMAND:-true}
```

Script

The script is sufficiently short to be shown. It uses the `basicLib.sh`, `moreLib.sh` and `comparisonLib.sh` libraries. The presence of the OPTIONS file is already checked.

```
 # Show the command that will be run to the student:

if $FW4EX_SHOW_SCRIPT
then
    fw4ex_show_script "$COMMAND"
fi

 # For every data directory, run student's and author's program and compare:

DATA_NUMBER=$( ls -d1 $DATA_DIR/*.${DATA_SUFFIX}/ | wc -l )
if [[ $DATA_NUMBER -eq 0 ]] ; then exit ; fi
cat <<EOF
<p>
Je vais comparer votre solution et la mienne sur $DATA_NUMBER
répertoires de données.
</p>
EOF

I=0
 # Iterate on every data directory:
for datadir in $DATA_DIR/*.${DATA_SUFFIX}/
do
    I=$(( $I+1 ))
    (
        trap 'echo "</section>"' 0
        OPTIONS=$(pwd)/$OPTIONS
        cd $datadir
        # Describe the test ie show the content of the data directory:
        echo "<section rank='$I'>"
        fw4ex_show_directory_content $I "$datadir"

        # Show student's command:
        fw4ex_show_student_command "$COMMAND" $COMMAND_FLAGS "< $COMMAND_STDIN"
        # Run student's command:
        fw4ex_run_student_command "$COMMAND" $COMMAND_FLAGS < $COMMAND_STDIN
        # and show raw result of this command:
        fw4ex_show_student_raw_result
        # Possible hook for authors:
        fw4ex_after_student_run_hook $I "$datadir"

        # Run author's command. If this command is erroneous, errors
        # will be emitted on stderr and sent to the author.
        fw4ex_run_teacher_command "$SOLUTION" $COMMAND_FLAGS < $COMMAND_STDIN
        fw4ex_show_teacher_raw_result
        fw4ex_after_teacher_run_hook $I "$datadir"

        # Normalize then show raw results:
        if [[ -n "$NORMALIZER" ]]
        then
            fw4ex_normalize_and_show_results
        fi
```

```
        # Compare normalized results and determine the win:
        fw4ex_compare_results
    )
done
```

8.10.5 Gap.sh

This pattern corresponds to an exercise that asks for a command. For instance, what command may convert the lower cases letters from the file mentioned in its first argument to upper case letters ?

This pattern gauges the student's answer by comparison to the author's solution. The command receives data file as argument, the result is in the stdout. To sum up and using the variables documented below, this pattern may be roughly characterized as:

```
$COMMAND $data
```

This pattern emits a grading report in French (encoded as UTF-8). Here is a short example:

```
NORMALIZER=removeTrailingBlanks
COMPARISON=line
TOTAL_WIN=10
WIN_FORMULA='triangular 0 0 5 $TOTAL_WIN/$DATA_NUMBER'

removeTrailingBlanks () {
   sed -e 's/ *$//'
}

source $FW4EX_LIB_DIR/Patterns/Gap.sh
```

Here are the parameters to configure this pattern.

COMMAND

This variable specifies the name of the file that contains the script to run. By default, this file is named `command`. This command will receive the name of the data file as an argument surrounded with LEFT_FLAGS and RIGHT_FLAGS.

```
COMMAND=${COMMAND:-command}
```

LEFT_FLAGS

This variable contains additional arguments to be given to `COMMAND` before the name of the data file. By default, this variable is empty.

```
LEFT_FLAGS=${LEFT_FLAGS:-}
```

RIGHT_FLAGS

This variable contains additional arguments to be given to `COMMAND` after the name of the data file. By default, this variable is empty.

```
RIGHT_FLAGS=${RIGHT_FLAGS:-}
```

COMMAND_STDIN

This variable specifies the standard input to be given to COMMAND. By default, this is /dev/null.

```
COMMAND_STDIN=${COMMAND_STDIN:-/dev/null}
```

SOLUTION

This variable specifies the name of the file that contains a correct solution that is a command achieving the desired behavior. By default, this pattern assumes that a perfect pseudo-job exists that contains the appropriate file named by the previous variable COMMAND.

```
SOLUTION=${SOLUTION:-$FW4EX_EXERCISE_DIR/pseudos/perfect/$COMMAND}
```

DATA_DIR

This variable is the name of the directory that contains the test files. By default, test files are located in the tests/ directory of the tar gzipped exercise. This directory should be defined with an absolute pathname.

```
DATA_DIR=${DATA_DIR:-$FW4EX_EXERCISE_DIR/tests}
```

DATA_SUFFIX

This variable tells the suffix the test files have. By default, the suffix is data. Any file with that suffix in the DATA_DIR directory is a test file. Test files are used in alphabetic order.

The number of tests is therefore the number of files in $DATA_DIR/*.$DATA_SUFFIX. This number will be computed by the pattern and set as the value of the DATA_NUMBER variable.

```
DATA_SUFFIX=${DATA_SUFFIX:-data}
```

NORMALIZER

This variable contains the name of the command (program + options or internal bash function as shown in the synopsis above) that normalizes the output of the programs to compare. You are not required to define this variable if you don't need normalization.

COMPARISON

This variable contains a word that defines the kind of comparison.

The char comparison computes the Levenshtein distance between the student's answer and the author's answer. The Levenshtein distance should not be used to compare too long strings.

The line comparison uses the diff utility to count the number of dissimilar lines.

The code comparison compares the exit codes of the student's and author's commands.

The int comparison compares the stdout of the student's and author's commands. These stdout are assumed to be integers.

The COMPARISON variable is there to ease switching from one comparison method to another. If you're sure you may remove that intermediate and patch the script below.

```
COMPARISON=${COMPARISON:-char}
```

TOTAL_WIN

This variable defines the maximal mark that may be won. Given the nature of the pattern, any test case allows the student to win TOTAL_WIN/DATA_NUMBER (rounded to two decimals only). To avoid embarrassing rounding, avoid a TOTAL_WIN of 1 with only 3 tests. The total mark will amount to 0.99 instead of 1.

WIN_FORMULA

This variable defines how to compute the number of points a student wins for one test file. The comparison (determined by COMPARISON) computes a distance: 0 is perfection (student's and author's normalized answers are the same). The WIN_FORMULA variable defines how to convert the distance into a mark. It defines the first arguments of a call to win.pl. See documentation of this utility for more details.

FW4EX_SHOW_SCRIPT

This boolean variable controls whether the student's script will be displayed or not. Byt default, the script is displayed.

```
FW4EX_SHOW_SCRIPT=${FW4EX_SHOW_SCRIPT:-true}
```

FW4EX_SHOW_COMMAND

This boolean variable controls whether to display the command to run.

```
FW4EX_SHOW_COMMAND=${FW4EX_SHOW_COMMAND:-true}
```

Script

The script is sufficiently short to be shown. It uses the basicLib.sh, moreLib.sh and comparisonLib.sh libraries. The presence of the COMMAND file is already checked.

```
 # Show the command that will be run to the student:

if $FW4EX_SHOW_SCRIPT
then
    fw4ex_show_script "$COMMAND"
fi

 # For every data file, run student's and author's program and compare:

DATA_NUMBER=$( ls -1 $DATA_DIR/*.${DATA_SUFFIX} | wc -l )
if [[ $DATA_NUMBER -eq 0 ]] ; then exit ; fi
cat <<EOF
<p>
Je vais comparer votre solution et la mienne sur $DATA_NUMBER fichier(s)
```

```
de données.
</p>
EOF

I=0
 # Iterate on every data file
for data in $DATA_DIR/*.${DATA_SUFFIX}
do
    I=$(( $I+1 ))
    (
        trap 'echo "</section>"' 0
        # Describe the test ie show the content of the data file:
        echo "<section rank='$I'>"
        fw4ex_show_data_file $I "$data"

        # Show student's command:
        fw4ex_show_student_command "$COMMAND" \
            $LEFT_FLAGS "$data" $RIGHT_FLAGS < $COMMAND_STDIN
        # Run student's command:
        fw4ex_run_student_command "$COMMAND" \
            $LEFT_FLAGS "$data" $RIGHT_FLAGS < $COMMAND_STDIN
        # and show raw result of this command:
        fw4ex_show_student_raw_result
        # Possible hook for authors:
        fw4ex_after_student_run_hook $I "$data"

        # Run author's command. If this command is erroneous, errors
        # will be emitted on stderr and sent to the author.
        fw4ex_run_teacher_command "$SOLUTION" \
            $LEFT_FLAGS "$data" $RIGHT_FLAGS < $COMMAND_STDIN
        fw4ex_show_teacher_raw_result
        fw4ex_after_teacher_run_hook $I "$data"

        # Normalize then show raw results:
        if [[ -n "$NORMALIZER" ]]
        then
            fw4ex_normalize_and_show_results
        fi

        # Compare normalized results and determine the win:
        fw4ex_compare_results
    )
done
```

8.10.6 Nice.sh

This pattern corresponds to an exercise that asks for a command. The command will be tested against sets of options contained in data files.

This pattern gauges the student's answer by comparison to the author's solution. The command receives the content of some data file as options, the result is in the stdout. To sum up and using the variables documented below, this pattern may be roughly characterized as:

```
$COMMAND $( cat $data )
```

This pattern emits a grading report in French (encoded as UTF-8). Here is a short example:

```
NORMALIZER=removeTrailingBlanks
COMPARISON=line
TOTAL_WIN=10
WIN_FORMULA='triangular 0 0 5 $TOTAL_WIN/$DATA_NUMBER'

removeTrailingBlanks () {
   sed -e 's/ *$//'
}

source $FW4EX_LIB_DIR/Patterns/Nice.sh
```

Here are the parameters to configure this pattern.

COMMAND

This variable specifies the name of the file that contains the script to run. By default, this file is named `command`.

```
COMMAND=${COMMAND:-command}
```

COMMAND_STDIN

This variable specifies the standard input to be given to `COMMAND`. By default, this is `/dev/null`.

```
COMMAND_STDIN=${COMMAND_STDIN:-/dev/null}
```

SOLUTION

This variable specifies the name of the file that contains a correct solution that is a command achieving the desired behavior. By default, this pattern assumes that a `perfect` pseudo-job exists that contains the appropriate file named by the previous variable `COMMAND`.

```
SOLUTION=${SOLUTION:-$FW4EX_EXERCISE_DIR/pseudos/perfect/$COMMAND}
```

DATA_DIR

This variable is the name of the directory that contains the test files. By default, test files are located in the `tests/` directory of the tar gzipped exercise. This directory should be defined with an absolute pathname.

```
DATA_DIR=${DATA_DIR:-$FW4EX_EXERCISE_DIR/tests}
```

DATA_SUFFIX

This variable tells the suffix the test files have. By default, the suffix is `data`. Any file with that suffix in the `DATA_DIR` directory is a test file. Test files are used in alphabetic order.

The number of tests is therefore the number of files in `$DATA_DIR/*.$DATA_SUFFIX`. This number will be computed by the pattern and set as the value of the `DATA_NUMBER` variable.

```
DATA_SUFFIX=${DATA_SUFFIX:-data}
```

NORMALIZER

This variable contains the name of the command (program + options or internal bash function as shown in the synopsis above) that normalizes the output of the programs to compare. You are not required to define this variable if you don't need normalization.

COMPARISON

This variable contains a word that defines the kind of comparison.

The `char` comparison computes the Levenshtein distance between the student's answer and the author's answer. The Levenshtein distance should not be used to compare too long strings.

The `line` comparison uses the `diff` utility to count the number of dissimilar lines.

The `code` comparison compares the exit codes of the student's and author's commands.

The `int` comparison compares the stdout of the student's and author's commands. These stdout are assumed to be integers.

The `COMPARISON` variable is there to ease switching from one comparison method to another. If you're sure you may remove that intermediate and patch the script below.

```
COMPARISON=${COMPARISON:-char}
```

TOTAL_WIN

This variable defines the maximal mark that may be won. Given the nature of the pattern, any test case allows the student to win `TOTAL_WIN/DATA_NUMBER` (rounded to two decimals only). To avoid embarrassing rounding, avoid a `TOTAL_WIN` of 1 with only 3 tests. The total mark will amount to 0.99 instead of 1.

WIN_FORMULA

This variable defines how to compute the number of points a student wins for one test file. The comparison (determined by COMPARISON) computes a distance: 0 is perfection (student's and author's normalized answers are the same). The `WIN_FORMULA` variable defines how to convert the distance into a mark. It defines the first arguments of a call to `win.pl`. See documentation of this utility for more details.

FW4EX_SHOW_SCRIPT

This boolean variable controls whether the student's script will be displayed or not. By default, the script is displayed.

```
FW4EX_SHOW_SCRIPT=${FW4EX_SHOW_SCRIPT:-true}
```

FW4EX_SHOW_COMMAND

This boolean variable controls whether to display the command to run.

```
FW4EX_SHOW_COMMAND=${FW4EX_SHOW_COMMAND:-true}
```

Script

The script is sufficiently short to be shown. It uses the `basicLib.sh`, `moreLib.sh` and `comparisonLib.sh` libraries. The presence of the COMMAND file is already checked.

```
 # Show the command that will be run to the student:

if $FW4EX_SHOW_SCRIPT
then
    fw4ex_show_script "$COMMAND"
fi

 # For every data file, run student's and author's program and compare:

DATA_NUMBER=$( ls -1 $DATA_DIR/*.${DATA_SUFFIX} | wc -l )
if [[ $DATA_NUMBER -eq 0 ]] ; then exit ; fi
cat <<EOF
<p>
Je vais comparer votre solution et la mienne sur $DATA_NUMBER jeux
d'options.
</p>
EOF

I=0
 # Iterate on every data file
for data in $DATA_DIR/*.${DATA_SUFFIX}
do
    I=$(( $I+1 ))
    (
        trap 'echo "</section>"' 0
        echo "<section rank='$I'>"

        # Show student's command:
        fw4ex_show_student_command "$COMMAND" $(cat "$data") "< $COMMAND_STDIN"
        # Run student's command:
        fw4ex_run_student_command "$COMMAND" $(cat "$data") < $COMMAND_STDIN
        # and show raw result of this command:
        fw4ex_show_student_raw_result
        # Possible hook for authors:
        fw4ex_after_student_run_hook $I "$data"

        # Run author's command. If this command is erroneous, errors
        # will be emitted on stderr and sent to the author.
        fw4ex_run_teacher_command "$SOLUTION" $(cat "$data") < $COMMAND_STDIN
        fw4ex_show_teacher_raw_result
        fw4ex_after_teacher_run_hook $I "$data"

        # Normalize then show raw results:
        if [[ -n "$NORMALIZER" ]]
        then
            fw4ex_normalize_and_show_results
        fi
```

```
        # Compare normalized results and determine the win:
        fw4ex_compare_results
    )
done
```

8.11 Languages

This section presents some libraries that are customized for a given programming language.

8.11.1 Java

Usually Java classes are tested with help of JUnit (version 3 or 4). JUnit (version 4.11) is present on the confined VM so you do not need to wrap this jar into your exercise (unless you do not want to depend on that precise version).

Some classes are provided: `org.fw4ex.junit.Assert`, `org.fw4ex.junit.TestRun` and `org.fw4ex.junit.ProgressiveTe`
If you statically import `org.fw4ex.junit.Assert` instead of the usual `org.junit.Assert` you get the same capabilities plus one: assertions are counted. This helps to grade a student since tests, assertions and failures are counted.

The second class `org.fw4ex.junit.TestRun` allows to run a series of test classes and emit, on its standard output stream, the resulting figure wrapped within three square brackets to make them easily parsable.

```
java -cp bin:$FW4EX_JUNIT_JAR org.fw4ex.junit.TestRun \
   org.fw4ex.junit.test.NumerousAssertionsTest
FW4EX JUnit version 4.11
..
Time: 0.011

OK (2 tests)


[[[103 Assertions, 3 Tests, 0 Failures]]]
```

Here follows the `org.fw4ex.junit.test.NumerousAssertionsTest` test class as an example:

```
// Copyright (C) Christian.Queinnec@upmc.fr
// GPL v2.

package org.fw4ex.junit.test;

/* Just a small class of JUnit tests with 2 tests and 102 assertions. */

import static org.fw4ex.junit.Assert.*;

import org.fw4ex.junit.Assert;
import org.fw4ex.junit.Fw4exResult;
import org.junit.*;

public class NumerousAssertionsTest {

        @BeforeClass
        public static void setup () {
                if ( null == Assert.getResult() ) {
                        Assert.setResult(new Fw4exResult(null));
```

```
                }
        }

        @Test
        public void test100assertions () {
                for ( int i=0 ; i<100 ; i++ ) {
                        assertTrue(true);
                }
                assertEquals(100, Assert.getResult().getAssertionCount());
                assertEquals(101, Assert.getResult().getAssertionCount());
        }

        protected int count = 0;
        @Test(expected=Throwable.class)
        public void testAbort () {
                if ( count++ < 1000 ) {
                        testAbort();
                } else {
                        count = 1/(count/1001 - 1);
                }
        }
}
```

However, `org.fw4ex.junit.TestRun` is not very robust since, if the output of a command is truncated then the last line will be lost. A more robust way is to use the third class, `org.fw4ex.junit.ProgressiveTestRun` as in:

```
% java -cp bin:$FW4EX_JUNIT_JAR org.fw4ex.junit.ProgressiveTestRun \
   -f $TMPDIR/lastResults.sh \
   org.fw4ex.junit.test.NumerousAssertionsTest
FW4EX JUnit version 4.11
..
Time: 0.011

OK (2 tests)
% cat $TMPDIR/lastResults.sh
ASSERTIONS=102
TESTS=2
FAILURES=0
```

The `-f` option allows to specify the file where the progress of the state of the test is recorded. This state is incremented for each test method, each assertion, each failure. Just source that file to get the three interesting variables.

There is a small difference on the number of assertions. `org.fw4ex.junit.TestRun` prints the number of checked assertions whereas `org.fw4ex.junit.ProgressiveTestRun` prints the number of attempted assertions. The two number are the same when there is no failure. Should some failures occur, then you must subtract FAILURES from ASSERTIONS to get the number of successfully checked assertions.

The path to the JUnit jar and these additional classes are held in variables `FW4EX_JUNIT_JAR` and `FW4EX_JAVA_BIN`.

8.11.2 Octave

See 8.12 for an example com.paracamplus.lt216.1 in Octave.

8.11.3 MzScheme

See 8.12 for an example org.fw4ex.li101.l2p in MzScheme.

8.12 Examples

Some examples of exercises (tar gzipped files) that may inspire authors are provided on the CodeGradX site. This section presents them and some of their characteristics so you may choose which one to start with.

bash org.example.li362.sh.7 This example uses the Gap pattern and asks for a little bash script.

bash org.fw4ex.li218.devoir.2010nov This exercise defines three separate questions and comes equipped with some accompanying files.

tr org.example.li362.tr.4 This example uses the BourgEnBresse pattern and asks for some options to give to the `tr` utility.

c org.example.li205.function.1 This is an example of an exercise asking for a C function.

java org.example.li314.java.3 This example compiles then tests a Java class that should implement a specified interface (this interface is available for the student). Tests are run using a refined version of JUnit.

mzscheme org.fw4ex.li101.l2p This example tests a Scheme function. This binding is very ugly and will be refined one day.

octave org.example.lt216.1 This example tests an Octave function. Fortunately Octave provides a sort of try-catch-finally making tests somewhat more robust.

Chapter 9

Campaign management

In order to compute some meaningful statistics, students must be grouped in cohorts and appreciated with respect to their cohort. Hence the concept of *campaign*.

A *campaign* is associated to

1. a selection of exercises,
2. a group of students,
3. a group of teachers,
4. a starting date
5. an ending date.

Campaigns are created by the CodeGradX administrators. The teachers of a campaign are the registered users belonging to the group of teachers of that campaign. Quite often, the students accessing a selection of exercises are adjoined to the students of the corresponding campaign.

9.1 Set of exercises

The Web servers displaying the set of exercises of a campaign, fetch that list from an X server. Teachers of that campaign can alter that list of exercises with new exercises or new version of former exercises.

`GET /exercisesset/path/ID`	return a JSON document describing a set of exercises
`PUT /exercisesset/path/ID`	set or replace a set of exercises with a JSON document
`POST /exercisesset/path/ID`	set or replace a set of exercises with a JSON document
`PUT /exercisesset/yml2json/ID`	set or replace a set of exercises with a YAML document
`POST /exercisesset/yml2json/ID`	set or replace a set of exercises with a YAML document

In the previous URLs, ID is the name of the set of exercises. Quite often, this is the name of the associated campaign.

The YAML format is the simpler, it will be compiled into a JSON document (not recommended since more difficult to write). Here is an example:

```
---
exercises:
  prologue: |
      Some exercises in Javascript.
      This is a multi-line text.
  1:
      title: Functions and Closures
      1: org.codegradx.js.min3.3
      2: org.codegradx.js.min4.2
  2:
    title: test
    epilogue: "fin d'exercices"
    1: 11111111-1111-1111-1112-000000000020
```

The YAML document defines an `exercises` key containing numeric sub-keys (starting from 1 and ending with the first missing numeric key). The numeric sub-keys may contain numeric sub-sub-keys associated to exercise (long) names or exercise UUID.

If an exercise long name is given, it refers to the last version of the exercise with that name at the time when the YAML document is converted into JSON. If an exercise UUID is given, it refers to a specific exercise.tgz.

Entries containing numeric keys should have an accompanying `title`. They may also have optional `prologue` and `epilogue` keys that is, sentences introducing and closing the list of exercises.

The previous example defines two sections of exercises, the first section contains two exercises with an explicit title, the second section contains only one exercise. It is also possible to avoid sections and to propose directly three exercises as in:

```
---
exercises:
  prologue: |
      Some exercises in Javascript.
      This is a multi-line text.
  title: functions
  1: org.codegradx.js.min3.3
  2: org.codegradx.js.min4.2
  3: 11111111-1111-1111-1112-000000000020
  epilogue: "fin d'exercices"
```

Chapter 10

Grading engine

This chapter describes how the `fw4ex.xml` is processed when a job is graded.

The grading engine uses UTF-8 everywhere. To avoid problems, your scripts should use the same encoding.

10.1 Exercise life-cycle

This section describes the life-cycle of an exercise. When submitted to the CodeGradX platform, an exercise passes through a series of steps described hereafter.

Archival An UUID is given to this new exercise.

Autocheck The pseudo-jobs contained in the exercise are graded in order to know if the exercise is coherent. This allows to detect a VM misfit where some programs required by the exercise are not present in the VM. A report is generated for the author and for the CodeGradX maintaineer.

Deployment If the autocheck step succeeds, the exercise will be deployed that is, made available for students.

There is a specific life-cycle when grading:

Installation The exercise targz is copied and inflated somewhere within the VM.

Initialization The exercise is initialized, that is the `initializing` element from the `fw4ex.xml` file is run under an author account in the `FW4EX_EXERCISE_DIR` directory that belongs to this author. It is not advised to create any file out of this directory (not even in `/tmp/`).

Grading After a successful initialization, the exercise is ready to process students' files (see next section for details).

Uninstallation After grading jobs, the exercise may be uninstalled that is, the HOME directory of the author account is entirely deleted. The exercise may be installed and uninstalled before and after each grading. It is up to the VM to choose (if it has enough room) to keep an installed exercise to grade future jobs.

10.2 Shells and streams

The grading element from the `fw4ex.xml` file is compiled into a shell script hereafter named `exercise.sh`. This compilation is a delicate balance involving shells, streams and confined scripts. Here follows a short description of the process.

First of all, the grading script (the `exercise.sh` file below) will be run under the identity of a student something like:

```
{ cat exercise.sh | su - student 3>out.xml 4>err.txt
} 1>fw4ex-outerr.txt 2>&1
```

If the grading script produces some results or warnings on its stdout and stderr, these will be sent to the CodeGradX maintaineer (since he is the writer of the compiler that produces that script). The `err.txt` file will gather the stderr produced by the author's scripts and will be sent back to the author. The `out.xml` is an XML file jointly produced by the grading script and the authors' script and will be sent to the student. This is an XML file and great care is taken to make it well formed and valid with respect to the CodeGradX grammar. If the `out.xml` is not valid, nothing will be sent to the student but the author will be notified of the problem in his `err.txt` file.

The grading script is made of three possible blocks. The first block is a prolog that defines some functions and variables. The other blocks might be script blocks or question blocks; they may appear more than once and they may be mixed at will. A question block is made of a prolog that defines some variables common to the question and of a number of script blocks. A script block is made of a prolog that defines some variables common to the block: some shell expressions that runs in a confined environment.

See Figure 10.1 for an example of a grading script.

```
prolog
script-block1
script-block2
question-block3
question-block4
script-block5
```

Figure 10.1: Grading script structure (example)

A sketch of a script block appears on Figure 10.2. The author script runs in a confined environment within a sub-shell, its stdout will be sent (after some massaging) to the student (in the `out.xml` file) while its stderr will be appended to the author's report (in `err.txt`). The exit code of the author's script is analysed. A non-null exit code may abort the whole exercise, the sole question or simply be ignored depending on the `iferror` attribute in the `script` element.

```
( some limits
  some environment settings
  confine author-script 1>out 2>err 3>/dev/null 4>/dev/null
) </dev/null
massage out 1>&3 2>>err
append err 1>&4
clean up temporary files out and err
react to confined author-script exit code
```

Figure 10.2: Script block structure

A sketch of a question block appears on Figure 10.3. A question knows its name from the environment. The files concerning the question and expected from the student are mecanically checked. Similarly to the core of a script block, the core of a question block is run in a sub-shell. The stdout and stderr of this subshell will be sent to the CodeGradX maintaineer. The stream 3 will be sent (after some massaging) to the

student (in the `out.xml` file) while stream 4 will be appended to the author's report (in `err.txt`). The exit code of a question block is analysed. A non-null exit code may abort the whole exercise or simply be ignored.

```
( FW4EX_QUESTION_NAME=q1
  check expectations
  some limits
  some environment settings
  script-block1
  script-block2
  script-block3
) </dev/null 3>out 4>err
massage out 1>&3 2>>err
append err 1>&4
clean up temporary files out and err
react to question exit code
```

Figure 10.3: Question block structure

The reaction of a script depends on whether it is run within a question or not. The specification *abort question* is non-sense when applied to a script that runs out of a question. The specification *abort exercise* is implemented as `exit 1`, the specification *abort question* when meaningful is implemented as `exit 0`, the specification *next script* is implemented as a no-op.

10.3 CodeGradX agent

A small server (the T server) is available for authors in order to autocheck their exercises. With this server, you upload an exercise (a targz file), it is autochecked and you get back a report telling you how are graded the various pseudo-jobs contained in the exercise. However, this is not conveniently automatizable. Therefore, a CodeGradX agent is provided in order to automatize the autochecking phase of an exercise and the grading of a batch of submissions. Here follow a series of examples of use of the CodeGradX agent.

ATTENTION: the CodeGradX agent is written in particularly dirty Perl code and is no longer maintained. A new agent, written in JavaScript, exists but requires Node.js to run: see 10.4.

10.3.1 Authenticating

In order to interact with the CodeGradX servers, you need to authenticate. You may then use the `--user` and `--password` options as in:

```
fw4ex-agent.pl --nosend --user=XXX --password=YYYY
```

However the `--password` option is dangerous since any other program running on the same machine may read it. Another, more secure, way is to provide this information via the `--credentials` option naming a file where login and password are stored. Be sure to protect this file!

```
fw4ex-agent.pl --nosend
```

The mentioned file is a YAML file (by default, this is `.fw4ex.yml` stored in the current directory) such as:

```
---
password: XXXXX
login: YYYYY
```

The `--update-credentials` updates this file and adds a line defining the cookie to use for further interactions. After the following command:

```
fw4ex-agent.pl --nosend --update-credentials
```

The file `.fw4ex.yml` then becomes:

```
---
login: YYYYY
password: XXXXX
cookie: UUUUUUUUUUUUUUUUUUUUUUUUUUUUU...@@
```

Authentication is performed with help from the X server. If you want to specify yourself the authentication server, use the `--xserver` and `--xprefix` options.

10.3.2 Obtaining exercises

If you want to act as as student and in order to submit jobs towards an exercise, you need to know the URL representing the exercise. CodeGradX does not let you guess these URL so you have to obtain them for instance via a `/path/PPP` request where PPP is the name of the set of exercises. Most often there is a dedicated (skinning) server for the set of exercises appropriate for a course.

```
fw4ex-agent.pl --type=path --path=li218 --report ./li218.xml
```

This command fetches the XML document (an `exercisesPath`) for the group of exercises named `li218` and stores it in the `li218.xml` file. The beginning of this document looks like (we split the long URL in order to fit line length):

```
<?xml version='1.0' encoding='utf-8' ?>
<fw4ex version='1.0' xml:lang='fr' lang='fr'>
 <exercisesPath name='li218'>
  <and>
   <set>
     <title>Autour de la commande <code>tr</code></title>
     <exercise exerciseid='11111111-1111-1111-2222-000000000001'
location='https://e.codegradx.org/exercisecontent/UA6tDmMIAG2DWXuUJqsD_DQ
 OySiRcMCP1hBJEZXkDksLqvTXZt7l5Crsx-NsUONRfhw5Z8Cs8mO95jscxvHc5VZ0GodaDlS6
 h13BRHUrtOWXJB78gtEuBFGkQnbiSr-hVcYNQylonhj4Ks2kAVeKK6ygxu9n8QnRY7mDC7teK
 81y15c2PN1SAt6ixUcHWWcZVpmiqd3AKt4YDfuiq27saVbyg0K68-P4UP2y2wxUDYENnjnJAE
 mGJVnVSv1vb9TflMKgrDkDSiZ7_0d1Zb4kNpUTJBS_zhjDTE64jBtxjFBROQAxAzciRJfd5K9
 8x9KSjJ07ik792Pdn7UIp_82fZTQ@@'>
      <identification name="org.example.li362.tr.1"
        nickname="majusculer" date="2008-08-27T14:52:00Z">
       <summary> Convertir en majuscules </summary>
       <tags>
        <tag name="UPMC"/>...
```

The `exercise` elements display the URL where may be found an exercise.

If the `--report` option is absent, the XML document is produced on the standard output.

By default, the CodeGradX agent uses the E server. You may change that with the `--eserver` and `--eprefix` options.

10.3.3 Posting a job

If you want to act as as student in order to submit a file to be graded may also be done with the CodeGradX agent. Let us suppose that we try to answer the exercise 11111111-1111-1111-2222-000000000001. We first store our answer in file `/tmp/options` (the name of the file is specified by the exercise) then we tar gzip our answer in `/tmp/options.tgz` then we post it (we split the long URL in order to fit line length).

```
echo -n '[0-9][0-9]*.txt' > /tmp/options
tar czf /tmp/options.tgz -C /tmp options
fw4ex-agent.pl --exercise 'UA6tDmMIAG2DWXuUJqsD_DQOySiRcMCP1hBJEZXkDksLqvTX
  Zt7l5Crsx-NsUONRfhw5Z8Cs8mO95jscxvHc5VZ0GodaDlS6h13BRHUrtOWXJB78gtEuBFGkQ
  nbiSr-hVcYNQylonhj4Ks2kAVeKK6ygxu9n8QnRY7mDC7teK81y15c2PN1SAt6ixUcHWWcZVp
  miqd3AKt4YDfuiq27saVbyg0K68-P4UP2y2wxUDYENnjnJAEmGJVnVSv1vb9TflMKgrDkDSiZ
  7_0d1Zb4kNpUTJBS_zhjDTE64jBtxjFBROQAxAzciRJfd5K98x9KSjJ07ik792Pdn7UIp_82f
  ZTQ@@' --stuff /tmp/options.tgz
```

This will return an XML document (a `jobStudentReport`) on the standard output, you may store it in a file with the `--report` option. This

```
<?xml version="1.0" encoding="UTF-8"?>
<fw4ex version="1.0">
 <jobStudentReport jobid="F8014F84-F7EF-11DF-8F78-B025982070F4">
  <marking archived="2010-11-24T17:26:27"
           started="2010-11-24T17:26:27Z"
           ended="2010-11-24T17:26:28Z"
           finished="2010-11-24T17:26:30"
           mark="0" totalMark="4">
   <machine nickname="Debian 4.0r3 32bit" version="1"/>
   <exercise exerciseid="F51F66D4F7EF11DF9440A90D982070F4"/>
  </marking>
  <report>
<FW4EX phase="begin" what="grading" when="2010-11-24T17:26:27Z"/>
<FW4EX what="initializing"/>
<FW4EX phase="begin" what="running script gCWYgae5s4"
       when="2010-11-24T17:26:27Z"/>
<p> Voici donc la commande que vous avez choisie:
<pre>
ls [0-9][0-9]*.txt
</pre></p>
<p>
Je vais comparer votre solution et la mienne sur 4

           ... Many elements omitted ...
<FW4EX what="exit code 0"/>
<FW4EX phase="end" what="question q3" when="2010-11-24T17:26:28Z"/>
<FW4EX phase="end" what="grading" when="2010-11-24T17:26:28Z"/>
  </report></jobStudentReport></fw4ex>
```

Grading an exercise may last a long time depending on the exercise (typically from 10 to 100 seconds). The CodeGradX agent busily waits for the report: some options may customize this waiting. The `--retry` option specifies how many times, the agent will try to download the final report. The `--offset` option specifies the number of seconds before asking for the report the first time. The `--timeout` option specifies the number of seconds to wait before two trials. Reasonable defaults exist for these options (5 for retry, 8 for offset, 3 for timeout).

By default, the CodeGradX agent uses the A server to post jobs and the S server to get the final report. You may change that with (respectively) the `--aserver` and `--aprefix` options and the `--sserver` and `--sprefix` options.

If you prefer an HTML report rather than an XML report, you have to pass through a skinning server such as `li218.codegradx.org`. You may specify your own skinning server with the `--skinserver` and `--skinprefix` options. The `--skin` option tells the CodeGradX agent that you want to use the default skinning server.

To copy-paste long URLs is painful therefore another syntax is available to specify an exercise. The URL `file:li218.xml#33` represents the 33rd exercise in the file `li218.xml`. Therefore to obtain the HTML report corresponding to the previous example, one may write:

```
fw4ex-agent.pl --exercise 'file:li218.xml#33' --stuff /tmp/options.tgz --skin
```

Sometimes, the retry, timeout and offset parameters are inappropriate. You must first be aware of the requests that are exchanged between the agent and the server (see chapter11). When you post a job, you receive a `jobSubmittedReport` XML answer from which you may know of the URL where you will get the `jobStudentReport`. Here is an example of a `jobSubmittedReport`:

```
<?xml version="1.0" encoding="UTF-8" ?>
<fw4ex version="1.0">
  <jobSubmittedReport
   location="/s/7/B/1/A/9/4/7/2/F/7/F/D/1/1/D/F/A/8/E/E/D/0/2/0/9/8/2/0/7/0/F/4">
    <job archived="2010-11-24T19:03:11"
         jobid="7B1A9472-F7FD-11DF-A8EE-D020982070F4" />
    <person personid="1000" />
    <exercise exerciseid="11111111-1111-1111-2222-000000000003" />
  </jobSubmittedReport>
</fw4ex>
```

To ease submission, instead of sending a targzipped file, you may just specify the directory to send and the CodeGradX agent will targzip the directory for you. Therefore, you may alternatively say (pay attention not to send huge amount of data such as `*~` files, `.svn` directories, etc.):

```
fw4ex-agent.pl --exercise 'file:li218.xml#33' --stuff /tmp/options/ --skin
```

You may restart the command to wait again for the `jobStudentReport` if you tell the CodeGradX agent where is the `jobSubmittedReport`. Intermediate answers from the CodeGradX servers are stored in the current directory (or another directory if you use the `--xmldir` option) under name `0.xml`, `1.xml` etc. (you may set the initial value of the counter with the `--counter` option). Therefore, when you post a job, you receive (by default) the `jobSubmittedReport` as `0.xml`. You may restart the command with:

```
fw4ex-agent.pl --type=jobcontent:0.xml --offset=0 --counter=10 --skin
```

Here we don't wait (offset is null), we reset the counter to 10 so the answer will not clobber the `0.xml` file and we still ask for an HTML report.

10.3.4 Posting multiple jobs

Sometime a teacher gets a number of students' files corresponding to a single exercise and wants to grade all of them in one go. It is possible to submit a batch of jobs. Let us suppose that we have two student's files in `/tmp/1.tgz` and `/tmp/2.tgz`.

```
mkdir /tmp/1 /tmp/2
echo -n '??*.txt' > /tmp/1/options
tar czf /tmp/1.tgz -C /tmp/1 .
echo -n '0-9*txt' > /tmp/2/options
tar czf /tmp/2.tgz -C /tmp/2 .
```

Since this is the teacher that submit the students' files and not the students themselves, a `multiJobSubmission` XML document is required to accompany the students' files. This document must list the targzipped files containing the students' files. Additionally, you may add a global label to identify this batch (this is particularly useful if you evolve the exercise and want to send this batch again). You are advised to add a label to each job. This label is important since this is the key, for the teacher, to relate a file to the concerned student. Note however that the CodeGradX platform only knows the requester of the batch (that is the teacher) but ignores who are the concerned students. These labels will be given back with the batch report so you may be able to process them and relate students to their reports.

```
cat > /tmp/fw4ex.xml <<EOF
<?xml version='1.0' encoding='UTF-8' ?>
<fw4ex version='1.0'>
  <multiJobSubmission label='batch.test.sh.1'>
    <job label='premiere copie, etudiant=1234'    filename='1.tgz' />
    <job label='seconde, { etudiant: 456}'        filename='2.tgz' />
  </multiJobSubmission>
</fw4ex>
EOF
```

When this XML file is created (check it against the CodeGradX RelaxNG schema), you may build the whole batch tar gzipped file and submit it with:

```
tar czf /tmp/batch1.tgz -C /tmp ./fw4ex.xml ./1.tgz ./2.tgz
fw4ex-agent.pl --type=batch --exercise 'file:li218#33' \
  --stuff /tmp/batch1.tgz --offset=30 --timeout=10
```

In this example, offset and timeout should be set according to the number of files to grade. If you set them too low and the CodeGradX agent stops working before the batch is completed, you may resume the agent. You must be aware of the requests that are involved in a batch submission see chapter 11). Suppose the XML file `0.xml` to contain the `multiJobSubmittedReport`, you resume the agent with:

```
fw4ex-agent.pl --type=batchcontent:0.xml
```

You will get in return a new XML document (a `multiJobStudentReport`) with the marks obtained by the students' files and the URLs of the associated grading reports. The CodeGradX agent may be used to fetch all the grading reports with the `--follow` option. Thus you may write

```
fw4ex-agent.pl --type=batch --exercise 'file:li218#33' \
  --stuff /tmp/batch1.tgz --follow
```

In this case, the students reports will appear as files stored in the local directory (or the directory specified by `--xmldir`) as `2.xml`, `3.xml`, etc. The `multiJobStudentReport` document tells you how far proceeds the CodeGradX platform, two attributes tell you the total number of files to grade and the total number of files already graded. You may resume the agent for instance with:

```
fw4ex-agent.pl --type=batchcontent:0.xml --offset=90 --timeout=60 \
  --follow --counter=100 --skin
```

Of course, you may also prefer HTML grading reports as shown in the previous example.

10.3.5 Posting a new exercise

Authors, that is, users of the CodeGradX platform blessed by CodeGradX maintaineers, may submit new exercises to the platform. If the exercise autochecks well then it will

be automatically deployed and the author receives the URL that identifies this exercise so he may build a site directing students towards this exercise.

To build an exercise, you must create a tar gzipped file containing all the necessary files describing the exercise that is, its stem, possibly some accompanying data files, some pseudo-jobs, the grading scripts and, finally, an XML descriptor binding all these. When you become an author, you get a personal prefix that you must use to name all the exercises you will create. This is typically something like `fr.lastname.firstname.`. Some prefixes are reserved to CodeGradX maintainers.

When posting the exercise (see the involved requests in chapter 11) you receive an XML document (an `exerciseSubmittedReport`) telling you where will pop up the `exerciseAuthorReport`. Here is an example of an `exerciseSubmittedReport`:

```
<?xml version="1.0" encoding="UTF-8"?>
<fw4ex version='1.0'>
  <exerciseSubmittedReport
     location='/s/F/5/1/F/6/6/D/4/F/7/E/F/1/1/D/F/9/4/4/0/A/9/0/D/9/8/2/0/7/0/F/4'
     jobid='F51F66D4-F7EF-11DF-9440-A90D982070F4' >
    <person personid='1000' />
    <exercise exerciseid='F51F66D4-F7EF-11DF-9440-A90D982070F4' />
  </exerciseSubmittedReport>
</fw4ex>
```

The final report (the `exerciseAuthorReport` XML document) will contain the URLs of the grading reports associated to the pseudo-jobs. Here is the beginning of such an XML document:

```
<?xml version="1.0" encoding="UTF-8"?>
<fw4ex version="1.0">
 <exerciseAuthorReport exerciseid="F51F66D4-F7EF-11DF-9440-A90D982070F4"
   safecookie="UhKv56QcMNhImwZmOz-WZ0uV2l1eg.....@@">
  <identification name="org.fw4ex.li218.devoir.2010nov"
   nickname="li218-devoir-2010nov" date="2010-10-29T19:46:06">
   <summary> Devoir LI218 de novembre 2010 </summary>
    <tags><tag name="UPMC"/><tag name="li218"/><tag name="shell"/></tags>
    <authorship>
     <author>
      <firstname>Christian</firstname>
      <lastname>Queinnec</lastname>
      <email>Christian.Queinnec@upmc.fr</email>
     </author>
    </authorship>
   </identification>
   <pseudojobs>
    <pseudojob jobid="F8014F84-F7EF-11DF-8F78-B025982070F4"
      location="/s/F/8/0/1/4/F/8/4/F/7/E/F/1/1/D/F/8/F/7/8/B/0/2/5/9/8/2/0/7/0/F/4"
      problem="1" duration="2">
     <submission name="null" expectedMark="0">
      <content directory="pseudos/null"/>
     </submission>
     <marking archived="2010-11-24T17:26:27"
              started="2010-11-24T17:26:27Z"
              ended="2010-11-24T17:26:28Z"
              finished="2010-11-24T17:26:30"
              mark="0" totalMark="4">
       <machine nickname="Debian 4.0r3 32bit" version="1"/>
       <exercise exerciseid="F51F66D4F7EF11DF9440A90D982070F4"/>
     </marking>
    </pseudojob>
   <!-- other pseudojobs omitted ... -->
```

```
   </pseudojobs><report/></exerciseAuthorReport></fw4ex>
```

Suppose the exercise be built as `exo.tgz`, then to submit it via the CodeGradX agent, write (after adjusting the offset, retry and timeout parameters if needed):

```
fw4ex-agent.pl --type=exercise --stuff exo.tgz \
  --offset=40 --follow --skin
```

Of course, if you did not get all the pseudo grading reports, you may ask the agent to resume its work (where `1.xml` is the `exerciseAuthorReport`) with:

```
fw4ex-agent.pl --type=exercisecontent:1.xml --follow --skin
```

When the exercise is autochecked successfully the `exerciseAuthorReport` contains the safecookie where it is deployed. If we suppose that `1.xml` holds this report then you may post a job towards this exercise with the following syntax (so you don't have to copy-paste the URL):

```
fw4ex-agent.pl --type=job --exercise 'file:1.xml' --stuff /tmp/options/ --skin
```

Options

The agent offers a number of options that might be tweaked. Here is the set of available options:

```
Can't locate WWW/Mechanize.pm in @INC (you may need to install the WWW::Mechanize module) (@INC c
BEGIN failed--compilation aborted at ../../Scripts/fw4ex-agent.pl line 183 (#1)
    (F) You said to do (or require, or use) a file that couldn't be found.
    Perl looks for the file in all the locations mentioned in @INC, unless
    the file name included the full path to the file.  Perhaps you need
    to set the PERL5LIB or PERL5OPT environment variable to say where the
    extra library is, or maybe the script needs to add the library name
    to @INC.  Or maybe you just misspelled the name of the file.  See
    "require" in perlfunc and lib.

Uncaught exception from user code:
        Can't locate WWW/Mechanize.pm in @INC (you may need to install the WWW::Mechanize module)
        BEGIN failed--compilation aborted at ../../Scripts/fw4ex-agent.pl line 183.
```

Conclusions

Typically, one may write the following in a Makefile in order to build, test an exercise and grade some jobs with this exercise. We suppose the tar gzipped files of the students to be in a `students/` directory. The `name.queinnec.essai.1` is the directory holding the exercise. Of course, you should set up the various options `--offset`, `--retry`, `--timeout` depending on the duration of the autocheck and batch processing.

```
AGENT=fw4ex-agent.pl
EXONAME=name.queinnec.essai.1
STUDENTS=students

all : ${EXONAME}.tgz post.exercise post.batch

${EXONAME}.tgz :
    xmllint --noout --relaxng fw4ex.rng ${EXONAME}/fw4ex.xml
    tar czf ${EXONAME}.tgz -C ${EXONAME} .

post.exercise : ${EXONAME}.xml
${EXONAME}.xml :
    ${AGENT} --nosend --update-credentials
```

```
    -rm ${EXONAME}.xml
    ${AGENT} --type=exercise --stuff ${EXONAME}.tgz --offset=90 --follow --skin
    grep safecookie 1.xml
    cp -p 1.xml ${EXONAME}.xml

batch.tgz :
    -rm -rf batch
    -rm -f batch.tgz
    mkdir batch
    echo "<?xml version='1.0' encoding='UTF-8'?><fw4ex version='1.0'> \
<multiJobSubmission label='$$(date)'>"    > batch/fw4ex.xml
        for tgz in $$(cd ${STUDENTS}; ls *.tgz) ; do \
                echo "<job label=\"$$tgz\" filename=\"$$tgz\" />" ; \
        done  >> batch/fw4ex.xml
        echo "</multiJobSubmission></fw4ex>" >> batch/fw4ex.xml
    xmllint --noout --relaxng fw4ex.rng batch/fw4ex.xml
    cp -p ${STUDENTS}/*.tgz batch/
    tar czf batch.tgz -C batch .

post.batch : ${EXONAME}.xml batch.tgz
    ${AGENT} --type=batch --exercise=file:${EXONAME}.xml \
      --stuff batch.tgz --counter=10 --follow --skin

get.batch.results :
    ${AGENT} --type=batchcontent:10.xml --offset=0 --follow --skin
```

In this `Makefile`, the labels of the jobs are not very useful unless the name of the students' files allow the teacher to determine to which student they belong.

10.4 The new CodeGradX agent in JavaScript

This new agent offers the same interface as the old agent in Perl. It requires Node.js (a JavaScript engine) to be present.

10.4.1 Installation

Use `npm` the Node Package Manager as in:

```
npm install -g codegradxagent
```

If you want to use the agent with the VM for authors, you need an additional NPM module: codegradxvmauthor. Install it with the following command and use the `codegradxvmauthor` command as the `codegradxagent` command.

```
npm install -g codegradxvmauthor
```

As all NPM modules, you'll find more information on the NPM site.

These two agents are supported by the `codegradxlib` and `codegradxenroll` NPM modules. These two modules are used for Web clients.

10.5 VM for authors

There is a Virtual Machine for authors that provides the full constellation of CodeGradX servers. You may run it to test and debug your exercises before submitting them to CodeGradX.

Chapter 11

XML formats

This chapter describes the XML formats that are used internally or externally. All XML files are wrapped within a `fw4ex` element and contain another unique element naming the kind of XML document. A single grammar named `fw4ex.rnc` (in RelaxNG compact form) rules them all.

The rest of this chapter is directly generated from `fw4ex.rnc` file.

11.1 Grammar

This grammar describes the content of all XML files read or generated by the FW4EX system. All these XML documents have a root element named `fw4ex` with a `version` attribute.

Where an attribute is optional, its default value is specified via an annotation belonging to the `annotation` namespace.

```
namespace annotation = "http://paracamplus.org/fw4ex/annotation/1.0"

start = fw4ex
```

There are a number of documents used for the various exchanges between students, teachers and servers. Students don't have to be aware of these documents. Authors should focus on the *exerciseSubmission* document that describes an exercise and, possibly, on the structure of the *jobStudentReport* generated by exercises. Deployers (teachers or web programmers) that want to connect their site to FW4EX should focus on the *exerciseContent* or *exerciseStem*, *jobStudentReport*.

jobSubmission

> This document is an internal document generated by an acquisition server when receiving some files. These files and this XML document named `fw4ex.xml` form a job that is, a tar gzipped file containing them all. The `fw4ex.xml` file gathers who is the student, what exercise is targeted, when the files were received. It also attributes an UUID to the job. This `fw4ex.xml` is packed with the files sent by the student (in a `content/` directory) to form the job.

jobSubmittedReport

> This document is the answer of an acquisition server when receiving a job submission. This acknowledgement is returned to the student. This document contains the information from the *jobSubmission* document. It also returns a `location` information that is, an URL where the grading report will appear. The `location` value is derived from the UUID christening the job.

exerciseSubmission

This document is an internal document generated when a fresh exercise is received by an exercise server. This XML document named `fw4ex.xml` gathers who is the author, when the exercise was received, if it is a new version of an old exercise. An UUID is given to the exercise that will follow the auto-checking phase of the exercise.

acquisitionServerState

This document describes the state of the acquisition server that is, it lists all the jobs that are waiting to be marked on the acquisition server. This is an internal document answered by the acquisition server to requests formed by administrative servers (and mainly the marking driver).

exerciseServerState

This document describes the state of the exercise server that is, it lists all the exercises that are present on the exercise server. This is an internal document answered by the exercise server to requests formed by administrative servers (and mainly the marking driver).

jobStudentReport

When a job is marked, the grading report is a `jobStudentReport` XML document. It is identified by the job UUID, it contains the text (the report) generated by the exercise and, finally, it contains a summary of the marking result (the mark, the total mark possible, the various dates when the report was generated).

jobAuthorReport

When a job is marked, it is possible that the programs of the author of the exercise generate anomalies. These anomalies may prevent the generation of the report to the student. These anomalies are gathered in a report and returned to the author to improve the exercise.

jobTrackerReport (FUTURE)

A tracker server is a server that tells where are stored the grading reports for students. More than one tracker may be requested. A tracker report may mention more than one storage server if some redundancy is wanted.

exerciseAuthorReport

When an exercise is submitted to the FW4EX system, an autocheck is run in order to determine if the exercise is well formed, complete and runs correctly. An exercise contains a number of pseudo-submissions that wil be graded. Their final mark is compared to the exected final mark. Any anomaly is returned to the author of the exercise and the exercise will not be deployed that is, not offered to students.

exercise

The concept of an exercise is the central piece of FW4EX. It is a fairly long text describing the many aspects of an exercise: what is the stem, the questions, what are the grading programs, where are the pseudo-copies, etc.

exerciseContent

An *exerciseContent* is an excerpt of an *exercise* corresponding to the whole set of information needed by a student to practice an exercise. It contains the stem, the data files, the expected content of the student's submission. Of course, it excludes the grading programs.

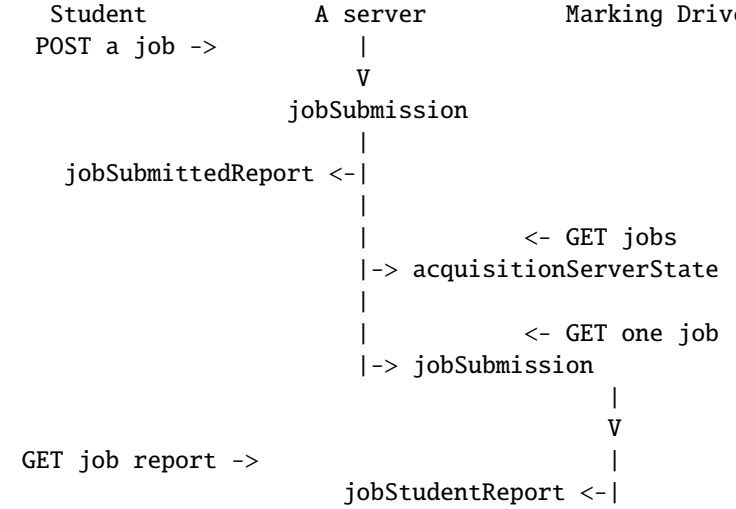
exerciseStem

An *exerciseStem* is an excerpt of an *exerciseContent* limited to the stem of the exercise. This document serves only for convenience for FW4EX clients that do not want to analyse an *exerciseContent* in order to extract the stem.

11.2 Use cases

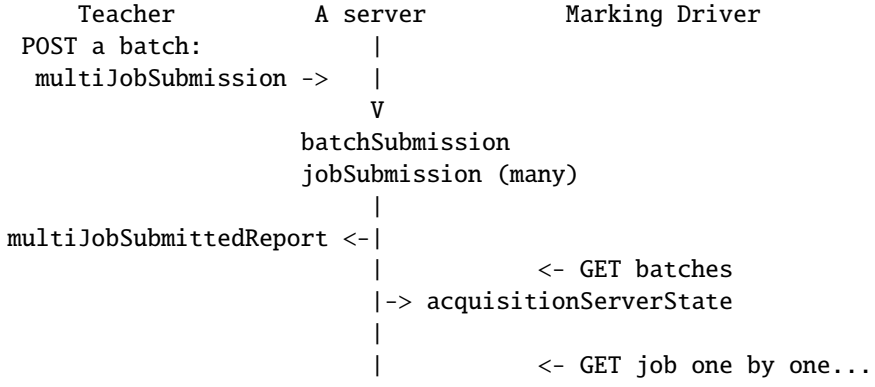
11.2.1 Student's submission

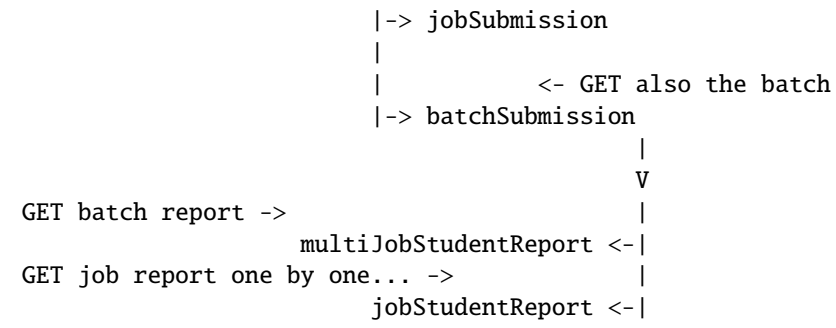
This is the use case where a student submits some files to be graded against one exercise. Only XML documents are shown. When receiving such a request, the A server elaborates a jobSubmission and responds with a jobSubmittedReport. Then marking drivers poll A servers, inspect acquisitionServerState, choose a waiting jobSubmission, process it and store the resulting jobStudentReport on the S server.



11.2.2 Teacher's batch submission

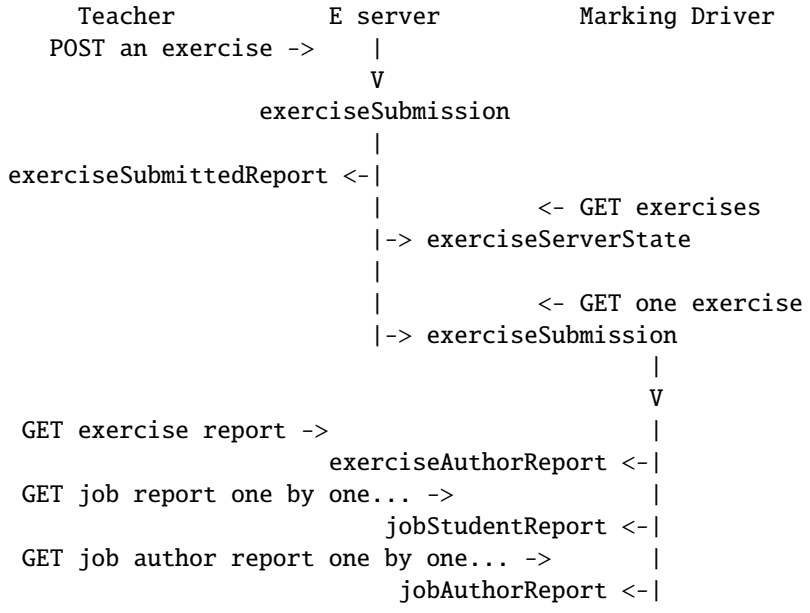
This is the use case where a teacher submits a batch of students' files. to be graded. Only XML documents are shown. The teacher generates a multiJobSubmission, the A server explodes this multiJobSubmission into multiple jobSubmissions, elaborates a batchSubmission and responds with a multiJobSubmittedReport. Marking drivers poll the A servers, get jobSubmission or batchSubmission and store their answers on the S server.





11.2.3 Teacher's exercise submission

This is the use case where a teacher submits an exercise. Only XML documents are shown. When an exercise is posted, the E server elaborates an exerciseSubmission document and answers with an exerciseSubmittedReport. Marking drivers poll E servers, choose an exercise to autocheck, get it, unwrap it and mark all the pseudojobs the exercise contains. If the exercise is successfully autochecked, it will be deployed.....



11.3 Root element: fw4ex

An `fw4ex` element has one mandatory attribute: the `version` attribute identifying the version of this grammar. Version numbers have a major.minor structure. Incompatible changes to this grammar increment the major number. Minor evolutions increment the minor number.

The optional `lang` and `xml:lang` attributes specify the language in which the exercise is written. French will use the values `fr` or `fr_FR` according to the usual standards.

```
fw4ex = element fw4ex {
   attribute version { "1.0" | "1.1" | "1.2" | "1.3" },
   # language of the exercise:
   attribute xml:lang { xsd:language } ?,
   attribute lang { xsd:language } ?,
```

```
    # Various kind of document:
    (  jobSubmission
     | jobSubmittedReport

     | multiJobSubmission
     | multiJobSubmittedReport
     | batchSubmission

     | exerciseSubmission
     | exerciseSubmittedReport

     | studentHistory
     | personHistory
     | exercisesList
     | exercisesPath
     | acquisitionServerState
     | exerciseServerState
     | jobsList
     | groupReport
     | authenticationAnswer
     | errorAnswer
     | constellationConfiguration

     | jobStudentReport
     | multiJobStudentReport

     | jobAuthorReport
     | exerciseAuthorReport

     | jobTrackerReport

     | exercise
     | exerciseContent
     | exerciseStem
  )
 }
```

11.4 jobSubmission

This is an internal document generated by an acquisition server to record a submission made by a student. This XML document will accompany the submitted files for further processing by the grading server. It contains three elements to describe the job, the student (cf. *person.id*) and the exercise (cf. *exercise.id*).

The element `job` contains two mandatory attributes: the `archived` attribute tells when the submission was recorded on the acquisition server, the `jobid` attribute is the UUID identifying the job.

```
jobSubmission = element jobSubmission {
   mixed {
      element job {
         # a possible label or comment:
         attribute label { xsd:string } ?,
         # date when the job was archived:
         attribute archived { xsd:dateTime },
         # The UUID identifying the job:
```

```
         attribute jobid { xsd:NMTOKEN },
         # An epoch when the corresponding stem was sent to the student:
         attribute stemdate { xsd:dateTime } ?
      },
      # The identifier of the student (an int from the Person table):
      person.id,
      # The identifier of the exercise (an UUID):
      exercise.id
   }
}
```

11.5 jobSubmittedReport

When a job is submitted, it is archived and a jobSubmissionReport is sent back to the student as acknowledgement. More precisely, this XML document is sent back to the FW4EX client software the student is using. This report contains the content of the jobSubmissionReport but includes an extra information: the `location` attribute that defines the URI where the *jobStudentReport* will appear. Note that this is an URI not an URL hence the client should know the storage server.

FUTURE: some hints about the possible (storage or tracker) servers may be given in the optional `servers` element.

```
jobSubmittedReport = element jobSubmittedReport {
   attribute location { xsd:anyURI },
   servers ?,
   mixed {
      element job {
         # date when the job was archived:
         attribute archived { xsd:dateTime },
         # The UUID identifying the job:
         attribute jobid { xsd:NMTOKEN }
      },
      # The identifier of the student (an int from the Person table):
      person.id,
      # The identifier of the exercise (an UUID):
      exercise.id
   }
}
```

11.6 multiJobSubmission

After an examination, a teacher may send, in one go, multiple students' submissions to the grading machine. These submissions form a 'batch'. They should be packed together in a single tar gzipped file with a mandatory accompanying `fw4ex.xml`. Very often the layout of the submitted tgz is: # ./fw4ex.xml ./students/1234567.tgz ./students/7891234.tgz ... # The accompanying `fw4ex.xml` gives some additional information that are completely useless for the grading machinery. However these information are useful for the teacher since they allow to tag the batch and the results in order to present derived information to students. This XML needs to be written by the teacher or some other tool the teacher uses to submit this batch.

```
multiJobSubmission = element multiJobSubmission {
```

```
   # A label given by the teacher to identify the batch. By default,
   # this is the time when the batch was submitted.
   attribute label { xsd:string } ?,
   # The (tgz) files to grade
   element job {
      # A label given by the teacher to identify the student (if
      # missing, the label will be equal to the filename). The meaning
      # of the label is only meaningful for the teacher, its semantics
      # is unknown from FW4EX.
      attribute label { xsd:string } ?,
      # the filename is a URl telling where one student's file is within the
      # whole tgz. Often, this is something such as C<students/1234567.tgz>
      attribute filename { xsd:string }
   } *
}
```

11.7 multiJobSubmittedReport

This is the acknowledgement sent to the teacher in response to a post of multiple submissions to grade. It only contains the id of the whole batch in order to get the associated report which will lead to the grading reports of the individual submissions contained in the batch.

```
multiJobSubmittedReport = element multiJobSubmittedReport {
   attribute location { xsd:anyURI },
   element batch {
      # date when the jobs were archived:
      attribute archived { xsd:dateTime },
      # The UUID identifying the whole batch of jobs:
      attribute batchid { xsd:NMTOKEN }
   },
   # The identifier of the submitter (an int from the Person table):
   person.id,
   # The identifier of the exercise (an UUID):
   exercise.id
}
```

11.8 batchSubmission

This is the XML file stored on an acquisition server that describes a batch of submissions to be graded. The final batch report will be available through an url built after `batchid`, the grading reports for the various submissions will be be available via the various `jobid`. This XML file is synthesized by an A server from the `multiJobSubmission` request.

```
batchSubmission = element batchSubmission {
   attribute label { xsd:string },
   attribute archived { xsd:dateTime },
   # The UUID identifying the whole batch of jobs:
   attribute batchid { xsd:NMTOKEN },
   # The identifier of the teacher who asked for this batch
   person.id,
```

```
   # The identifier of the exercise (an UUID):
   exercise.id,
   element job {
      attribute label { xsd:string },
      attribute jobid { xsd:NMTOKEN }
   } *
}
```

11.9 exerciseSubmission

This is an internal document generated by an exercise server when receiving a new exercise. This document will be packed with the files sent by the author, it records the author (cf. *person.id*) and contains some information in attributes: when the exercise was archived and the UUID attributed to this exercise.

An optional element may specify the UUID of a previous exercise. The submission should then refer to the same exercise, the submission is therefore a new version of the exercise. This additional element may only be used by administrators.

```
exerciseSubmission = element exerciseSubmission {
   attribute location { xsd:anyURI },
   mixed {
      element job {
         # date when the job was archived:
         attribute archived { xsd:dateTime },
         # The UUID identifying the job:
         attribute jobid { xsd:NMTOKEN }
      },
      # The identifier of the author (the requester):
      person.id,
      exercise.id
   }
}
```

11.10 exerciseSubmittedReport

This document is returned to an author after submitting an exercise. The `location` field contains the URL where the report will be stored after autochecking the exercise.

```
exerciseSubmittedReport = element exerciseSubmittedReport {
   attribute location { xsd:anyURI },
   attribute jobid { xsd:NMTOKEN },
   # The int identifying a person:
   person.id,
   # The UUID identifying the exercise:
   exercise.id
}
```

11.11 studentHistory

This report lists the jobs concerning a student.

```
studentHistory = element studentHistory {
   # The identifier of the student:
   attribute personid { xsd:positiveInteger },
   attribute lastname { xsd:string },
   attribute firstname { xsd:string },
   attribute pseudo { xsd:string },
   # The list of jobs
   element job {
      attribute jobid { xsd:NMTOKEN },
      # date when the job was archived on server A:
      attribute archived { xsd:dateTime },
      attribute mark { xsd:decimal },
      attribute totalMark { xsd:decimal },
      attribute _href { xsd:anyURI },
      # The identifier of the exercise:
      exercise.id,
      empty
   } *
}
```

11.12 personHistory

This report lists the jobs, exercises and batches concerning a person.

```
personHistory = element personHistory {
   # The identifier of the person:
   attribute personid { xsd:positiveInteger },
   attribute lastname { xsd:string },
   attribute firstname { xsd:string },
   attribute pseudo { xsd:string },
   [ annotation:default = "false" ]
   attribute author { xsd:boolean } ?,
   # The list of jobs
   element jobs {
      element job {
         attribute jobid { xsd:NMTOKEN },
         # date when the job was archived on server A:
         attribute archived { xsd:dateTime },
         attribute mark { xsd:decimal },
         attribute totalMark { xsd:decimal },
         attribute _href { xsd:anyURI },
         # The identifier of the exercise:
         exercise.id,
         empty
      } *
   } ?,
   # the list of exercises
   element exercises {
      element exercise {
         attribute exerciseid { xsd:NMTOKEN },
         attribute name { xsd:string },
         attribute nickname { xsd:string },
         attribute start { xsd:dateTime },
```

```
         attribute _href { xsd:anyURI },
         empty
      } *
   } ?,
   # the list of batches
   element batches {
      element batch {
         attribute batchid { xsd:NMTOKEN },
         attribute label { xsd:string },
         attribute archived { xsd:dateTime },
         attribute _href { xsd:anyURI },
         exercise.id,
         empty
      } *
   } ?
}
```

11.13 exercisesList DEPRECATED in favor of exercises-Path

This element lists a series of exercises. It tells the exercises a student may choose among.

```
exercisesList = element exercisesList {
   element exercise {
      attribute exerciseid { xsd:NMTOKEN },
      attribute location { xsd:anyURI },
      identification ?
   } *
}
```

11.14 exercisesPath

This element describes an ordered series of exercises as recommended by a teacher. Among the set of exercises, some are mandatory, others are suggested. One may also mixes some text to comment the path.

```
exercisesPath = element exercisesPath {
   attribute name { xsd:NMTOKEN },
   exercisesPathItem
}

exercisesPathItem =
   exercisesPathItemAnd
 | exercisesPathItemOr
 | exercisesPathItemSet
 | exercisesPathItemNone
 | exercisesPathItemExercise
 | exercisesPathItemComment

exercisesPathItemExercise = element exercise {
   attribute exerciseid { xsd:NMTOKEN },
```

```
    attribute location { xsd:anyURI },          # to be removed
    attribute uuid { xsd:NMTOKEN } ?,           # to be made mandatory (same as exerciseid ???)
    identification ?
=cut
 }
 exercisesPathItemNone = element none {
    empty
 }
 exercisesPathItemComment = element comment {
    xhtml.inline.text
 }
 exercisesPathItemOr = element or {
    element title { xhtml.inline.text } ?,
    element prologue { xhtml.inline.text } ?,
    exercisesPathItem +,
    element epilogue { xhtml.inline.text } ?
 }
 exercisesPathItemAnd = element and {
    element title { xhtml.inline.text } ?,
    element prologue { xhtml.inline.text } ?,
    exercisesPathItem +,
    element epilogue { xhtml.inline.text } ?
 }
 exercisesPathItemSet = element set {
    element title { xhtml.inline.text } ?,
    element prologue { xhtml.inline.text } ?,
    exercisesPathItem +,
    element epilogue { xhtml.inline.text } ?
 }
```

11.15 constellationConfiguration (FUTURE)

This document gives information on the available servers and their roles within the FW4EX constellation. Normally any server of the constellation may answer that document so a client may discover the other servers of the constellation.

```
 constellationConfiguration = element constellationConfiguration {
    server +
 }

 servers = element servers {
    server +
 }

 server = element server {
    attribute type { 'acquisition' | 'exercise' | 'storage' | 'tracker' },
    attribute name { xsd:NMTOKEN },
    attribute priority { xsd:nonNegativeInteger } ?,
    attribute urlprefix { xsd:anyURI } ?,
    element comment {
       text
    } ?
 }
```

11.16 jobTrackerReport (FUTURE)

A tracker server tells on which server(s), the client may find a precise grading report (given its URI). The tracker server returns an ordered list of possible servers.

```
jobTrackerReport = element jobTrackerReport {
   attribute location { xsd:anyURI },
   servers
}
```

11.17 acquisitionServerState

This document describes the state of an acquisition server. The document is dated (with the clock of the server). The `number` attribute specifies how many jobs exist on the acquisition server waiting to be graded. The `number` attribute corresponds to the number of elements `job` that are children of the `acquisitionServerState` element.

```
acquisitionServerState = element acquisitionServerState  {
   # when this request was served:
   attribute date { xsd:dateTime },
   # number of archived jobs (see next tags):
   attribute number { xsd:nonNegativeInteger },
   # As much jobs as specified in the preceding 'number' attribute:
   ( element job {
      # date when the job was archived:
      attribute archived { xsd:dateTime },
      # The UUID identifying the job:
      attribute jobid { xsd:NMTOKEN },
      # How many times this job has been served by an A server:
      attribute served { xsd:nonNegativeInteger }
   } | element batch {
      # date when the batch was archived:
      attribute archived { xsd:dateTime },
      # The UUID identifying the batch:
      attribute batchid { xsd:NMTOKEN }
   } ) *
}
```

11.18 exerciseServerState

This document lists all the fresh exercises stored in the exercise server that need to be autochecked.

```
exerciseServerState = element exerciseServerState {
   # when this request was served:
   attribute date { xsd:dateTime },
   # number of archived exercises (see next tags):
   attribute number { xsd:nonNegativeInteger },
   # As much exercises as specified in the preceding 'number' attribute:
   element exercise {
      # date when the exercise was archived:
      attribute archived { xsd:dateTime },
      # The UUID identifying the exercise:
```

```
      attribute exerciseid { xsd:NMTOKEN }
   } *
}
```

11.19 jobsList

This element lists a series of jobs related to an exercise.

```
jobsList = element jobsList {
   exercise.id,
   element job {
      attribute jobid { xsd:NMTOKEN },
      # date when the job was archived on server A:
      attribute archived { xsd:dateTime },
      attribute waitduration { xsd:decimal } ?,
      attribute markduration { xsd:decimal } ?,
      attribute totalduration { xsd:decimal } ?, # wait + mark durations
      attribute mark { xsd:decimal },
      attribute totalMark { xsd:decimal },
      attribute _href { xsd:anyURI },
      element person {
         attribute personid { xsd:positiveInteger },
         attribute lastname { xsd:string },
         attribute firstname { xsd:string }
      }
   } *
}
```

11.20 authenticationAnswer

This message is used by an authentication server as an answer to a successful authentication. It is also used as an answer to the registration of a new person in order to describe what is in the database. It the user is an author, also returns the prefix that he may use to name the exercices he authors.

```
authenticationAnswer = element authenticationAnswer {
   element person {
      attribute personid       { xsd:positiveInteger },
      attribute expirationDate { xsd:dateTime } ?,
      attribute lastname       { xsd:string } ?,
      attribute firstname      { xsd:string } ?,
      attribute pseudo         { xsd:string } ?,
      attribute email          { xsd:string } ?,
      attribute accesslink     { xsd:string } ?,
      element author {
         attribute prefix { xsd:string }
       } *
   }
}
```

11.21 groupReport

This document lists the students of a group and some of their characteristics. Presently, the skill is an integer between 0 and 100 (100 being the better). If the requester is not an admin, only pseudoes are given not real names.

```
groupReport = element groupReport {
  attribute synthetized { xsd:dateTime },
  attribute groupName { xsd:NMTOKEN },
  element person {
     attribute personid { xsd:positiveInteger },
     attribute lastname { xsd:string } ?,
     attribute firstname { xsd:string } ?,
     attribute pseudo { xsd:string },
     attribute level { xsd:positiveInteger }
  } *
}
```

11.22 groupsReport

This document lists all groups. This requires to be admin.

```
groupsReport = element groupsReport {
  attribute synthetized { xsd:dateTime },
  element group {
    attribute groupName { xsd:NMTOKEN }
  } *
}
```

11.23 errorAnswer

This message is used whenever some problem is detected. The `person` element may be present if the user is correctly authenticated.

```
errorAnswer = element errorAnswer {
   element person {
      attribute personid { xsd:positiveInteger },
      attribute name { xsd:string },
      attribute expirationDate { xsd:dateTime }
   } ?,
   #element request {
   #   xsd:string         # hint about the request              FUTURE ???
   #} ? ,
   element message {
      attribute code { xsd:string },
      element reason { xsd:string }
   }
}
```

11.24 jobStudentReport

This document is a the grading report generated by FW4EX. The `jobid` attribute identifies the job, the `marking` element sums up the main information synthetized by the

grading engine, the final element report contains the (potentially lengthy) text report. This text is written with a XHTML-like syntax.

```
jobStudentReport = element jobStudentReport {
   # The UUID identifying the job:
   attribute jobid { xsd:NMTOKEN },
   marking,
   element report { xhtml.content }
}
```

11.25 multiJobStudentReport

Sometimes, a teacher may want to grade a number of submissions in one go: this is specified by a multiJobSubmission element and acknowledged with a multiJobSubmittedReport. When the submissions are graded, a multiJobStudentReport is returned. This document tells where are the individual student reports.

```
multiJobStudentReport = element multiJobStudentReport {
   attribute batchid { xsd:NMTOKEN },
   attribute archived { xsd:dateTime },
   attribute label { xsd:string } ?,
   # number of entirely graded jobStudentReports:
   attribute finishedjobs { xsd:nonNegativeInteger },
   # total number of jobs to be graded:
   attribute totaljobs { xsd:nonNegativeInteger },
   element jobStudentReport {
      attribute label { xsd:string } ?,
      attribute jobid { xsd:NMTOKEN },
      attribute location { xsd:anyURI },
      [ annotation:default = "0" ]
      attribute problem { "0" | "1" } ?, # default 0
      element marking {
         attribute started { xsd:dateTime },
         attribute finished { xsd:dateTime },
         attribute mark { xsd:decimal },
         attribute totalMark { xsd:decimal }
      }
   } +
}
```

11.26 jobAuthorReport

If the programs (contained in an exercise) grading a job produce errors (on stderr) then this stderr is wrapped into a report in order to be analysed by the author of the exercise. The report element contains a text since, in presence of anomalies, it is not wise to expect a valid xml fragment. The marking element sums up the main information synthetized by the grading programs as far as they work.

```
jobAuthorReport = element jobAuthorReport {
   # The UUID identifying the job: This UUID allows the author to get
   # the associated student report:
   attribute jobid { xsd:NMTOKEN },
   element marking {
```

```
      attribute archived { xsd:dateTime },
      # date when the VM starts marking the job:
      attribute started { xsd:dateTime },
      # date when the VM ends marking the job:
      attribute ended { xsd:dateTime },
      # the precise marker that graded the job:
      machine,
      # The identifier of the exercise:
      exercise.id
   },
   # an unstructured text for authors or fw4ex maintaineer:
   element report { text }
}
```

11.27 exerciseAuthorReport

When an author submits an exercise, the exercise is autochecked that is, all the pseudo-jobs it contains are graded. This document gathers the jobStudentReports and jobAuthorReports for all these pseudo-copies.

The `exerciseid` attribute is an UUID christening the exercise. The `identification` element is a copy of the one given by the author in the exercise. The `pseudojobs` container contains a sequence of `pseudojob` elements. Each of them contains an attribute `jobid` to identify the generated job for that occasion, a copy of the corresponding `submission` element from the exercise and the `marking` element that sums up the information synthetized by the grading engine.

A general text might be produced in the `report` element, to gather the anomalies detected in the descriptor of the exercise (its `fw4ex.xml` file). Some parts may be missing if the exercise is badly conditioned (no `fw4ex.xml` file for instance).

```
exerciseAuthorReport = element exerciseAuthorReport {
   attribute exerciseid { xsd:NMTOKEN },
   # This attribute is only present when the exercise had been
   # successfully autochecked. This is a safe cookie allowing the
   # author to use the freshly autochecked exercise.
   attribute safecookie { xsd:string } ?,

   identification ?,

   element pseudojobs {
      # number of entirely graded jobStudentReports:
      attribute finishedjobs { xsd:nonNegativeInteger },
      # total number of (pseudo-)jobs to be graded:
      attribute totaljobs { xsd:nonNegativeInteger },
      element pseudojob {
         # The jobid gives access to the student's and author's reports:
         attribute jobid { xsd:NMTOKEN },
         attribute location { xsd:anyURI },
         [ annotation:default = "0" ]
         attribute problem { "0" | "1" } ?, # default 0
         attribute duration { xsd:positiveInteger } ?, # in seconds
         submission,
         marking ?,
         # An unstructured text with evidences of problems
         element report { text } ?
```

```
      } *
   } ?,

   # An unstructured summary text
   element report { text } ?
}
```

11.28 exercise

This document describes an exercise and its various facets. Here follows the meaning of the great sections composing the description of an exercise.

identification

This section identifies the exercise, its version, its authors.

conditions

This section describes the financial (how much to pay) and technical (which OS, which language, which proficiency, etc.) conditions associated to the exercise.

equipment

This section describes the files that accompany the exercise, they should be sent to the student. These may be examples, documentations, data files, etc.

initializing

This section describes what must be done to prepare a student's machine before he may work on an exercise or to prepare a grading machine before it may grade a job.

content

This section describes the questions composing the exercise.

autochecking

This section defines the pseudo-jobs that is, the non-regression tests to determine if the exercise is well deployed.

grading

This section defines how to grade a job.

```
exercise = element exercise {
   identification,
   conditions,
   equipment ?,
   initializing ?,
   content,
   autochecking,
   grading
}
```

11.29 exerciseContent

When a student wants to practice an exercise, its fw4ex-enabled client receives an extract of the content of the exercise that is, the questions and the accompanying files. Grading procedures and other critical information are not sent. These files are sent in a zipped file containing an `fw4ex.xml` file describing the content of the zipped file. This XML document is an `exerciseContent` element defined as follows.

The `synthesisDate` attribute is the date when the zipped file was created.

```
exerciseContent = element exerciseContent {
   # Creation date of this exerciseContent:
   attribute synthesisDate { xsd:dateTime },

   identification,
   conditions,
   equipment ?,
   content,

   characteristics ?
}
```

11.29.1 characteristics

This element contains numbers extracted from the database. These numbers may be used to help users to select exercises or to help the client runtime to make the user wait for the report.

```
characteristics = element characteristics {
   element statistics {
      # Mean time to process a job (extracted from the db):
      attribute meantime { xsd:decimal },
      # Mean number of attempts to succeed with the exercise:
      attribute meantrials { xsd:decimal },
      # Number of students having attempted to do this exercise:
      attribute students { xsd:nonNegativeInteger },
      # Number of students that succeeded:
      attribute sucesses { xsd:nonNegativeInteger }
   }
}
```

11.30 exerciseStem

Some fw4ex-enabled clients (the javascript browser version for instance) prefer to receive selected parts of the previous zip file. The `exerciseStem` contains the displayable content of the exercise that is, the introduction and the questions.

```
exerciseStem = element exerciseStem {
   # Creation date of this exerciseStem:
   attribute synthesisDate { xsd:dateTime },
   #   some urls ???
   #   exercise.id, ???
   identification,
   equipment ?,
   content
}
```

11.31 content

The content element contains an optional introduction, a sequence of questions followed by an optional conclusion. The introduction and conclusion element may be an XHTML inlined text or refer to an external file (in the tar gzipped exercise) containing this XHTML text.

An exercise always have at least one question. It may have only one question for one-liner exercises for instance.

```
content = element content {
   # The maximal mark that can be obtained with this exercise:
   # This is a possible annotation that, if present, should be coherent
   # with the sum of questions' totalMarks.
   attribute totalMark { xsd:decimal } ?, # CHECK! only positive floats!
   # a longer text serving as an introduction to the exercise
   element introduction {
      infile.or.inline.xhtml.content
    } ?,
   content.question +,
   element conclusion {
      infile.or.inline.xhtml.content
    } ?
}
```

11.32 content.question

A question element is identified by an internal name (used for internal references: this name is used to get the associated grading programs). The totalMark attribute determines the maximal mark that might be given when grading this question. The sum of the totalMark of all questions sets the total mark that might be obtained when grading the whole exercise.

The stem element contains an XHTML-like inlined text asking a question or may refer to an external file holding this XHTML-like text. The external file is a file from the tar gzipped exercise.

The expectations element is a container defining the files (and their structuring directories) that are expected in a student's submission.

The other elements hint and solution are reserved for some future, they are not implemented for now.

The hint element defines a text that might appear after a given duration. This text may help a student to find his way towards the solution.

The solution (not sent in exerciseStem document of course) may contain a solution that might be used (or displayed) by a grading program if useful.

```
content.question = element question {
   # All question names must have a different name:
   attribute name { xsd:NMTOKEN },
   # A human-readable title instead of the previous (short) name:
   attribute title { string } ?,
   # The maximal mark that can be obtained with this question:
   attribute totalMark { xsd:decimal }, # CHECK! only positive floats!
   # files expected from the student (their name is imposed):
   element expectations {
      # Are all expectations listed ?
```

```
      attribute exhaustive { xsd:boolean } ?,
      # What to do in case of missing expectations:
      attribute iferror {
          "abort exercise"
        | "abort question"
      } ?,
      expectation *
   },
   # The text of the question:
   element stem {
      infile.or.inline.xhtml.content
   },
   # Maybe some hints that will appear later...             NYI
   element hint {
      attribute when { xsd:duration }, # in seconds
      infile.or.inline.xhtml.content
   } *,
   element solution {
      infile.or.inline.xhtml.content
      # and some additional resources or URLS towards explanations ???
   } ?
}
```

11.33 infile.or.inline.xhtml.content

In many places where texts are expected, it is possible or to put the text in the appropriate XML element or to store it in a separate file somehere in the exercise tar gzipped file. For small texts, the first solution might be preferred but it augments the size of the `fw4ex.xml` exercise description. The second solution potentially leads to many small files but these small files may be shared by different exercises and may therefore factor some common texts.

To refer to a separate file, use the `authorfilename` attribute otherwise insert the text in the content of the element. Filenames are specified in Unix notations that is, with slashes to express directory structures. Conventionnally, the filename do not start with a slash. For example, if the `someExercise.tgz` file contains

```
fw4ex.xml
data/a.txt
stem/Q1.xml
```

Then to refer to the `Q1.xml` file, one should write:

```
authorfilename='stem/Q1.xml'
```

CHECK what happens when the filename starts with a slash ???

```
infile.or.inline.xhtml.content =
   (
      xhtml.content
    | # relative to ~author/
      # CHECK! No leading / please! No funny chars!
      attribute authorfilename { xsd:string }
   )
```

11.34 autochecking

The autochecking element defines how the exercise is checked before being offered to students. This element contains submission elements corresponding to submissions whose expected mark will be checked.

```
autochecking = element autochecking {
   submission +
}
```

11.35 submission

The submission element defines the files that a student may submit. These files will then be graded and the final mark should be in accordance with the expected mark. This allows to check that the grading programs work well, that the virtual machine contains all the utilities needed to grade.

A submission has a name so it may report anomalies with the name of the submission. Usual names are null, perfect, almost etc. I usually add new submissions after fixing grading bugs to be sure I've fixed them!

The epsilon attribute is there to compensate the rounding problem. All marks are rounded up to two decimals so, to assert that 0.99 and 1 are close enough, just set epsilon to be greater than 0.01.

A submission with a true skip attribute must not be marked. The associated pseudo submission is not yet ready.

The submission.content defines the content of the submission.

```
submission = element submission {
   attribute name { xsd:Name },
   # The copy must be graded with a mark equal to expectedMark +/- epsilon
   attribute expectedMark { xsd:decimal },
   [ annotation:default = "0.01" ]
   attribute epsilon { xsd:decimal } ?,
   [ annotation:default = 'false' ]
   attribute skip { xsd:boolean } ?,
   # The content of the submission:
   submission.content
}
```

11.36 submission.content

The submission may be given inline or be contained in an external directory.

```
submission.content = element content {
   submission.external.content
 | submission.inline.content
}
```

11.37 submission.external.content

If the submission is contained in a directory then mention that directory. Conventionally, submissions are in a sub-directory (named after the name of the submission) of the pseudos/ directory. For instance, an exercise tgz might be:

```
fw4ex.xml
pseudos/null/
pseudos/perfect/program
```

In which case, the XML fragment might be:

```
<submission name='perfect' expectedMark='20' directory='pseudos/perfect'/>
<submission name='null' expectedMark='0'directory='pseudos/null'/>
```

NOTE: empty directories are somewhat problematic in tar or zip archives. It is better to create a empty file within them.

```
submission.external.content =
   # relative to ~author/
   attribute directory { xsd:string },
   empty
```

11.38 submission.inline.content

When the files are only small texts, they may be specified inline in the exercise description. The `basename` is the name of the file, the `trim` attribute specifies if leading and trailing spaces or newlines should be removed. The content of the file might be given in the `content` attribute or as content of the `file` element. Therefore,

```
<file basename='foo.txt' content='Hello World'/>
```

is the same as:

```
<file basename='foo.txt' trim='yes'/>
     Hello World
</file>
```

```
submission.inline.content =
   element file {
      attribute basename { xsd:Name },
      attribute trim { "yes" | "no" } ?,
      ( text
       | (
          attribute content { xsd:string } &
          empty
         )
      )
   } +
=cut
```

11.39 marking

The `marking` element sums up the main results of the grading process. The `archived` attribute specifies when the job was posted by the student. The `started` attribute specifies when the VM started grading the job, the `ended` attribute specifies when the VM ended grading the job. The `finished` attribute specifies when the student and author's reports were made available to students or authors.

The `mark` attribute is the mark given by the grading engine, the `totalMark` is a copy of the maximal mark that might be given for that exercise.

The machine element specifies which machine graded the job, the exercise.id identifies which exercise (mainly which version) was used to grade the job.

Eventually, if the exercise contains several questions, the mark of every question appears in the partialMark element paired with the name of the question.

```
marking = element marking {
   # date when the job was archived on server A:
   attribute archived { xsd:dateTime },
   # date when the VM starts marking the job:
   attribute started { xsd:dateTime },
   # date when the VM ends marking the job:
   attribute ended { xsd:dateTime },
   # date when the markengine finishes storing results:
   attribute finished { xsd:dateTime },
   attribute mark { xsd:decimal },
   attribute totalMark { xsd:decimal },
   # the precise marker that graded the job:
   machine ?,
   # The identifier of the exercise:
   exercise.id,
   # marks per question
   element partialMark {
      attribute name { xsd:NMTOKEN },
      attribute mark { xsd:decimal }
   } *
}
```

11.40 initializing

The initializing section defines how to prepare the student machine in order to be able to practise the exercise. It also defines how to prepare the grading engine to be able to grade a job. These actions may be: compile a library, uncompress some data files, etc. These actions are specified by scripts.

```
initializing = element initializing {
   script +
}
```

11.41 grading

The grading element defines how to grade a student's submission. It first defines which machine should be used, the limit to set, the POSIX environment to set up then a series of scripts to run.

```
grading = element grading {
   # The maximal mark that can be obtained with this exercise:
   attribute totalMark { xsd:decimal } ?, # CHECK! only positive floats!
   machine,
   # how should be graded every question:
   limit *,
   environment ?,
   ( grading.question | command ) +
}
```

11.42 machine

The machine element specifies the VM required to mark the student's submission. There are some predefined VM but you may specify your own. You may also specify the version number of the machine you want to use though upward compatibility is a goal that is, a new machine should not grade differently the jobs graded by an old version.

```
machine = element machine {
   # The nickname of the virtual machine to use (for instance a Debian
   # 4.0r3 32bits)                          FUTURE Here ? identification ?
   (
      attribute nickname { xsd:string },
      attribute version { xsd:nonNegativeInteger } ? )
   |  attribute name { xsd:string }
}
=cut
```

11.43 grading.question

A grading.question specifies how to check a question. The question is referred to by its name (see the name attribute of the question element in the content element. The commands to run may be limited (see limit) and benefit from some POSIX variables (see environment).

If the attribute enabled if present and equal to yes, the question will not be graded. This attribute allows the author to test only a part of a multi-questions exercise.

```
grading.question = element question {
   # Reference the associated question (described in the 'terms' section):
   attribute name { xsd:NMTOKEN },
   [ annotation:default = "yes" ]
   attribute enabled { "yes" | "no" } ?,

   limit *,
   environment ?,
   command +
}
```

11.44 limit

Limits include timeout, cpu, diskio, etc. The name of these limits are predefined (according to 'man bash'). The nicknames for the limits may also be used (they are defined in /etc/security/limits.conf).

Some limits may specify the unit. Others don't. For example,

```
<limit predefined='stack' value='10' unit='Mi'/>
<limit predefined='nice' value='5'/>
<limit predefined='cpu time' value='10' unit='seconds'/>

limit = element limit {
   attribute predefined {
      "core file size"             #   (blocks, -c) 0
    | "core"
    | "data seg size"              #   (kbytes, -d) unlimited   # Of course not
```

```
     | "data"
     | "max nice"                  #           (-e) 20
     | "nice"
     | "file size"                 #   (blocks, -f) unlimited
     | "fsize"
     | "pending signals"           #           (-i) unlimited
     | "sigpending"
     | "max locked memory"         #   (kbytes, -l) unlimited
     | "memlock"
     | "max memory size"           #   (kbytes, -m) unlimited
     | "rss"
     | "open files"                #           (-n) 1024
     | "nofile"
     | "pipe size"                 #(512 bytes, -p) 8
     | "POSIX message queues"      #    (bytes, -q) unlimited
     | "msgqueue"
     | "max rt priority"           #           (-r) unlimited
     | "rtprio"
     | "stack size"                #   (kbytes, -s) 8192
     | "stack"
     | "cpu time"                  #  (seconds, -t) unlimited
     | "cpu"
     | "max user processes"        #           (-u) unlimited
     | "nproc"
     | "virtual memory"            #   (kbytes, -v) unlimited
=cut
     | "file locks"                #           (-x) unlimited
     | "locks"
    },                             # where block = 1024 bytes.
    attribute value { xsd:nonNegativeInteger },
    attribute unit {
       "block"    | "blocks"
     | "byte"     | "bytes"
     | "second"   | "seconds"
     | "M"  | "k"  # absolute numbers: 10^6 and 10^3.
     | 'Mi' | 'ki' # absolute numbers: 2^20 and 2^10.
    } ?
 }
```

11.45 environment

These elements introduce or remove POSIX variables into or from the environment. They may introduce in the context of the exercise, a question or a single script. The scope of the variable is accorded.

```
 environment = element environment {
    ( environment.assignment
    | environment.hide
    ) +
 }
```

11.46 environment.assignment

This element introduces a POSIX variable. These variables are useful for the author and should not disturb the FW4EX engine therefore no variable with a prefix of FW4EX is allowed. The variable may be specified with a value or a pathname targeting the author directory.

```
<set name='WHAT' value='42'/>
<set name='FILE' authorfilename='data/some.file'/>
```

In the last example, the value of FILE will the be the absolute filename leading to the file `data/some.file` from the exercise tgz.

```
environment.assignment = element set {
   attribute name { xsd:NMTOKEN - ("^FW4EX.*") },
   ( attribute value { xsd:string }
   | # relative to ~author/
     attribute authorfilename { xsd:string }
   )
}
```

11.47 environment.hide

This element specifies which POSIX variable(s) to hide from the confined program. The variable may be specified by its name or a set of variables may be specified by a regular expression.

```
environment.hide = element hide {
   ( attribute name { xsd:NMTOKEN }
   | attribute regexp { xsd:NMTOKEN }     # NOT YET IMPLEMENTED
   )
}
```

11.48 command

A command may be predefined or may refer to a script.

11.49 predefined.action

Currently, there is only one predefined action: the `echo` action (reminiscent of the similar task from Ant).

11.50 echo

Instead of writing a script to emit a string, something like:

```
<script>
   cat <<EOF
<p>Hello <em>you</em></p>
EOF
</script>
```

One may write alternatively one of the following:

```
<echo><p>Hello <em>you</em></p></echo>
<echo message="<p>Hello <em>you</em></p>"/>
<echo authorfilename='hello.you'/>
```

Where `hello.you` is a file (in the exercise targz) containing some text.
BUG: UNICODE letters seem to be translated into Latin1 ???

```
echo = inline.echo | attributed.echo | external.echo
inline.echo = element echo {
   xhtml.inline.text
 | xhtml.enumeration
 | xhtml.paragraph
}
attributed.echo = element echo {
   attribute message { xsd:string },
   empty
}
external.echo = element echo {
   # relative to ~author/
   attribute authorfilename { xsd:string },
   empty
}
```

11.51 script

A script is a series of commands written in some scripting language (sh, perl, ocaml, etc.). The content of the script may be specified inline (within the XML element) or in some external file. If the script node has a `idref` attribute then it is generated from another node (the one with the associated `id` attribute). This accomodates the fact that nodes whose content is written in the `fw4exsh` language is compiled into `bash`. It is up to the marking slave to run the compiled version or to interpret the original source. Of course, the other version has to be ignored (hence the id-ref link).

```
script = inline.script | xml.script | external.script
```

11.52 common.script.content

Whether inlined or externally defined, scripts share a number of common characteristics. Scripts may be limited, the environment may be altered, the behaviour after an error may also be specified.

iferror

If the script exits with an erroneous exit code (a byte different from zero) then either the entire grading process may be aborted, either the grading process of the current question is aborted or nothing occurs (this is the default action) and the grading process resumes with the next script.

iferror = attribute iferror { "abort exercise" | "abort question" | "next script" }

limit

There are two kinds of limits that might be set. The limits inherited from the `ulimit` POSIX command or, more finely, the limits accepted by the `confine` utility which are three:

= over

maxcpu

This tells how many seconds the script is allowed to run. This is a wall-clock duration therefore the script might be impacted if the grading machine is busy.

maxout

This tells how many bytes the script is allowed to produce on its stdtout. You may use the multiplier `k` (1000), `M` (1000*1000) or `ki` (1024) or `Mi` (1024*1024).

maxerr

This tells how many bytes the script is allowed to produce on its stdterr. You may use the multiplier `k` (1000), `M` (1000*1000) or `ki` (1024) or `Mi` (1024*1024).

```
common.script.content =
   limit *,
   environment ?,
   # What to do in case of problem (i.e., exit value != 0):
   [ annotation:default = "next script" ]
   iferror ?,
   # parameters for confiner:
   attribute maxcpu { xsd:nonNegativeInteger } ?,
   attribute maxout { xsd:NMTOKEN { pattern = "\d+([kM]i?)?" } } ?,
   attribute maxerr { xsd:NMTOKEN { pattern = "\d+([kM]i?)?" } } ?,
   # arguments for the script:                NOT YET IMPLEMENTED
   argument *
```

11.53 inline.script

An inline script is specified in the body of the `script` element. By default, the script is assumed to be a `sh` script. The `trim` attribute removes leading and trailing spaces.

The script should be runnable that is, may start with a `#!` comment specifying the interpreter to run. This first line will be added if the `language` attribute is present and no such first line already exists.

Pay attention to XML and avoids using less-than signs without precaution. To ease readability, instead of writing:

```
<script>
   read w &lt; some.file
</script>
```

It is preferrable to write:

```
<script><![CDATA[
   read w < some.file
]]></script>
```

```
inline.script = element script {
   common.script.content,
   language.attribute ?,
   attribute trim { "yes" | "no" } ?,
   text
}
```

11.54 xml.script

Instead of writing shell scripts you may generate them with a graphical UI. The GUI stores its state in XML, a restricted subset of shell in XML syntax named `fw4exsh`. For now, we suppose that if an `xml.script` node is present then an `inline.script` is also present with the compiled version. The GUI uses the first node to restaure its state but must regenerate accordingly the other node in case of changes.

11.54.1 script

This `fw4exsh` language describes the structure of a marking script. This is a tree of loops with commands as leaves. The `chDir` command allows to change the current directory.

Many of these nodes may be annotated with a `totalMark` attribute stating what is the maximal number of points that might be won after evaluation of this node.

```
script_element =
   loop
 | chDir
 | command
```

11.54.2 loop

It is possible to loop over an enumeration of strings or to loop over a set of files (or directories). Loops are named so it is possible to refer to them in order to know the number of iteration, the current index and the current value.

```
loop = loopOnFiles | loopOnStrings
```

11.54.3 chDir

Changes the current directory. This change is limited to the evaluation of the body.

```
chDir = element chDir {
   (
      # to refer to the current value of the index of the loop of that name:
      attribute nameref { xsd:NMTOKEN }
    | (
       attribute fw4exdir { "/" | "teacher" | "student" },
       attribute dirname { xsd:string }
      )
   ),
   element body { script_element }
}
```

11.54.4 command

Commands may represent an assertion (checking whether some property hold) or the comparison of some student's program output with teacher's output.

```
command = assertCommand | compareCommand
```

11.54.5 component

A command is compound of the name of a program to run and some `components` specifying with which command line arguments, which input streams, etc.

11.55 external.script

This element specifies a program to run. This program may be in the targz exercise file or in the common library.

```
external.script = element script {
   common.script.content,
   (
      # relative to ~author/
      attribute authorfilename { xsd:string }
    |
      # relative to FW4EX_LIB_DIR/         # NOT YET IMPLEMENTED
      attribute scriptname { xsd:string }
   ),
   # FUTURE: maybe some arguments ?
   empty
}
```

11.56 argument (NOT YET IMPLEMENTED)

Some general scripts may require arguments to be tailored to a specific task. Arguments must be given in positional order, they may be regular strings or filenames relative to the exercise targzipped file or relative to the student's files.

```
argument = element argument {
   ( attribute value { xsd:string }
   |
      # file relative to ~student/
      attribute studentfilename { xsd:string }
   | # relative to ~author/
      attribute authorfilename { xsd:string }
   )
}
```

11.57 expectation.directory

An `expectation.directory` element defines the basename of the directory. A comment (an XHTML-like text) may be associated. This comment may appear in an interactive FW4EX client to hint what this directory is for. The `all` attribute tells whether the entire content of the directory should be submitted. If `all` is true the inner expectations must also be checked.

```
expectation.directory = element directory {
   attribute basename { xsd:string },
   [ annotation:default = "false" ]
   attribute all { xsd:boolean } ?,
   element comment { xhtml.inline.text } ?,
   expectation *
}
```

11.58 expectation.file

An `expectation.file` element describes a file that the student should submit. A comment (an XHTML-like text) may be associated. This comment may appear in an interactive FW4EX client to hint what this file should contain. An `initial` element may contain hints about the height (in lines) and width (in columns) of a widget that might be used to collect the student's input. The initial content of the widget might as well be specified.

```
expectation.file = element file {
   attribute basename { xsd:string },
   element comment { xhtml.inline.text } ?,
   [ annotation:default = "lf" ]
   attribute eol { "lf" | "cr" | "crlf" } ?,
   [ annotation:default = "UTF-8" ]
   attribute coding { "UTF-8" | "ISO-8859-1" } ?,
   [ annotation:default = "mandatory" ]
   attribute presence { "mandatory" | "optional" } ?,
   attribute show { xsd:boolean } ?,
   # The shape of the solution (may be used to prefill the widget that
   # will contain the student's solution). Attributes are hints for the
   # number of lines of the expected solution.
   element initial {
      attribute height { xsd:positiveInteger } ?,
      attribute width { xsd:positiveInteger } ?,
      text
   } ?
}
```

11.59 equipment.content

If the `a/b.c` and `a/d.e` files must be sent then this will be described as:

```
<directory basename='a'>
  <file basename='b.c'/>
  <file basename='d.e'/>
</directory>
```

or, alternatively, as:

```
<directory basename='a'>
  <file basename='b.c'/>
</directory>
<directory basename='a'>
  <file basename='d.e'/>
</directory>
```

```
equipment.content = ( file | directory ) *
```

11.60 file (PARTIALLY IMPLEMENTED)

This element describes a file. Only the `basename` is a required attribute. Among the others maybe the `eol` attribute will be implemented to cope with end-of-lines for text files.

When file is part of the equipment, the comment may be used by a FW4EX-client to accompany a link to get the file. When file is part of the expectations, the comment may be used to accompany an input box or file input box.

```
file = element file {
   attribute basename { xsd:Name },
   element comment { xhtml.inline.text } ?,
   attribute size { xsd:nonNegativeInteger } ?,
   attribute digest { xsd:NMTOKEN } ?,
   attribute digestAlgorithm { "sha1" } ?,
   [ annotation:default = "binary" ]
   attribute type { "text" | "binary" } ?,
   [ annotation:default = "lf" ]
   attribute eol { "lf" | "cr" | "crlf" } ?,
   [ annotation:default = "application/octet-stream" ]
   attribute mimetype { xsd:string } ?,
   [ annotation:default = "false" ]
   attribute hidden { xsd:boolean } ?
}

directory = element directory {
   attribute basename { xsd:Name },
   ( file | directory ) *
}
```

11.61 tag

Exercises may be tagged with names (usually short names that is, words). These tags may stress the type of exercise (examination, one-liner, etc.), the language of the answer (C, Java, bash, sed, etc.), the set of exercises comprising this exercise, etc.

```
tag = element tag {
   attribute name { xsd:Name }
}
```

11.62 authorship

This element defines who are the authors, how to communicate with them, related information describing them. Their contribution to the exercise may also be described. Authors are identified by their email though internally (in the database), authors are, like person, identified by an integer.

```
authorship = element authorship {
   element author {
      attribute since       { xsd:dateTime } ?,
      attribute till        { xsd:dateTime } ?,
      element firstname     { xsd:string },
      element middlename    { xsd:string } ?, # may be a simple initial
      element lastname      { xsd:string },
      element postlastname  { xsd:string } ?, # additional postfixed names
      # This email is used to identify the author:
      element email         { xsd:string },
      # This email is used by students to communicate directly with the author:
```

```
      element exerciseEmail { xsd:string } ?,
      element siteurl       { xsd:anyURI } ?,
      element comment       { xhtml.inline.text } ?
   } +,
   element contributor {
      attribute since       { xsd:dateTime } ?,
      attribute till        { xsd:dateTime } ?,
      element firstname     { xsd:string },
      element middlename    { xsd:string } ?, # may be a simple initial
      element lastname      { xsd:string },
      element postlastname  { xsd:string } ?, # additional postfixed names
      # This email is used to identify the contributor:
      element email         { xsd:string },
      # This email may be used by persons
      element exerciseEmail { xsd:string } ?,
      element siteurl       { xsd:anyURI } ?,
      element comment       { xhtml.inline.text } ?
   } *
 }
=cut
```

11.63 conditions

This element specifies under which conditions this exercise may be practised. This element defines the cost (in Euro). A description describes the resources needed to practice the exercise: these conditions may be on the student's machine OS, or required libraries or required skills, etc.

```
 conditions = element conditions {
    attribute cost { xsd:double },
    attribute costunit { "euro" },
=cut

    # This description is shown to the student and describes the machine,
    # the OS, the languages, the libraries needed for the exercise:
    element description { xhtml.content }
 }
```

11.64 Common abbreviations

These are common abbreviations used in this grammar.

```
 exercise.id = element exercise {
    attribute exerciseid { xsd:NMTOKEN },
    attribute safecookie { xsd:string } ?,
    attribute name { xsd:string } ?,
    attribute nickname { xsd:string } ?,
    attribute totalMark { xsd:decimal } ?
 }
 person.id = element person {
    attribute personid { xsd:positiveInteger }
 }
 job.id = element job {
```

```
   attribute jobid { xsd:NMTOKEN }
}
```

11.65 xhtml.section

A section has a title and a body. A `name` may be specified for internal links. The `rank` attribute may be used to number the sections.

```
xhtml.section = element section {
   attribute name { xsd:NMTOKEN } ?,
   attribute rank { xsd:nonNegativeInteger } ?,
   xhtml.title ?,
   xhtml.content
}

xhtml.title = element title {
   xhtml.inline.text
}

xhtml.paragraph =
   xhtml.text.paragraph
 | xhtml.codeblock
 | xhtml.image
 | fw4ex.warning
 | fw4ex.error
 | fw4ex.success
```

11.65.1 image PROVISIONAL

Sometimes, it might be useful to embed an image within the grading report.

```
xhtml.image = element img {
   attribute src { xsd:string },
   attribute width { xsd:positiveInteger } ?,
   attribute height { xsd:positiveInteger } ?,
   attribute alt { xsd:string } ?,
   empty
}
```

11.65.2 xhtml.text.paragraph

Be rather loose: Accept p elements within a p element.

```
xhtml.text.paragraph = element p {
   (
      xhtml.inline.text
    | xhtml.codeblock
    | xhtml.text.paragraph
    | fw4ex.warning
    | fw4ex.error
    | fw4ex.success
   ) +
}
```

11.66 warning

A warning may be emitted to notify a weird situation but that does not require to stop the grading engine. For instance, a light error may be corrected 'en passant' by the grading engine but notified.

```
fw4ex.warning = element warning {
   (
      xhtml.inline.text
    | xhtml.codeblock
   ) +
}
```

11.67 error

This element is used to notify an error to the student.

```
fw4ex.error = element error {
   (
      xhtml.inline.text
    | xhtml.codeblock
   ) +
}
```

11.68 success

This element is used to notify a success to the student.

```
fw4ex.success = element success {
   (
      xhtml.inline.text
    | xhtml.codeblock
   ) +
}
```

11.69 xhtml.codeblock

This element is used to present some code. A special stylesheet may address these elements.

In order to present an interaction between a machine and a user, one may distinguish the two with the machine and user elements. Here is an example:

```
<pre>
<machine>% </machine><user> date
</user><machine>Thu Dec 25 15:13:30 CET 2008
% </machine></pre>
```

NOTE: No newline character between machine and user tags within a <pre> element.

```
xhtml.codeblock = element pre {
   # useful for <pre id='fw4ex_student_code'>
   attribute id { xsd:NMTOKEN } ?,
   attribute data-language { xsd:NMTOKEN } ?,
```

```
   mixed {
    (  xhtml.code.user
     | xhtml.code.machine
     | xhtml.code.line.number
     | fw4ex.anchor
    ) *
   }
}

xhtml.code.user = element user {
   text
}
xhtml.code.machine = element machine {
   text
}
xhtml.code.line.number = element lineNumber {
   text
}
```

11.70 xhtml.enumeration

As usual there are numbered and unnumbered enumerations.

```
xhtml.enumeration =
   xhtml.ordered.enumeration
 | xhtml.unordered.enumeration

xhtml.ordered.enumeration = element ol {
   ( element li { xhtml.inline.text }
    | xhtml.codeblock
   ) +
}

xhtml.unordered.enumeration = element ul {
   ( element li { xhtml.inline.text }
    | xhtml.codeblock
   ) +
}
```

11.71 xhtml.inline.text

This element defines a text that appears within a single paragraph. These text fragments may be styled as in HTML, they may contain a partial mark stating the the student wins a number of points or they may contain additional information (`fw4ex.anchor`) for the sole needs of the grading platform.

```
xhtml.inline.text = mixed {
   ( xhtml.styled
   | xhtml.code
   | fw4ex.partial.mark
   | fw4ex.anchor
   ) *
}
```

11.72 fw4ex.partial.mark

This element states that the student wins `value` points. The same `value` appears as the body of the element so it may be styled with some CSS. To be valid, the partial mark must contain a valid key known by the author but not by the student.

```
fw4ex.partial.mark = element mark {
   attribute key { xsd:NMTOKEN },
   attribute value { xsd:decimal },
   xsd:decimal
}

xhtml.styled =
   xhtml.emph
 | xhtml.bold
 | xhtml.sub
 | xhtml.sup
 | xhtml.anchor
 | fw4ex.warning
 | fw4ex.error
 | fw4ex.success
 | xhtml.normal
```

11.73 xhtml.comparison (NOT YET IMPLEMENTED)

May be used in some future when student's and teacher's texts must be compared. Some javascript may be used to stress the differences.

```
xhtml.comparison = element comparison {
   element student { xhtml.paragraph },
   element teacher { xhtml.paragraph }
}
```

11.74 xhtml.file.annotation

This element gathers annotations with respect to a student's file. An annotation has a kind (a short word telling which type of annotation it is. From this an icon may also be inferred) and an associated text.

Annotations annotate part of the student's file. They may be hooked at a precise location (specified by a line and a column) or be associated to a region of the file.

```
xhtml.file.annotation = element annotations {
   attribute studentfilename { xsd:string },
   xhtml.annotation *,
   # an overall comment for the whole file:
   xhtml.inline.text ?
}

xhtml.annotation = xhtml.line.annotation | xhtml.region.annotation

xhtml.line.annotation = element annotation {
   attribute kind { xsd:NMTOKEN },
   attribute line { xsd:nonNegativeInteger },
```

```
   attribute column { xsd:nonNegativeInteger },
   xhtml.inline.text
}
xhtml.region.annotation = element annotation {
   attribute kind { xsd:NMTOKEN },
   attribute start-line { xsd:nonNegativeInteger },
   attribute start-column { xsd:nonNegativeInteger },
   attribute stop-line { xsd:nonNegativeInteger },
   attribute stop-column { xsd:nonNegativeInteger },
   xhtml.inline.text
}
```

11.75 fw4ex.anchor

These elements are reserved for the FW4EX platform. They are used to comment the grading process and to tidy up the generated xhtml. This is often useful since bash lacks a try-catch-finally feature so it is difficult to ensure that all opening tags do have their associated closing tags.

```
fw4ex.anchor = element FW4EX {
   attribute phase { "begin" | "end" } ?,
   attribute what { xsd:string },
   attribute when { xsd:dateTime } ?,
   empty
}
```

11.76 Final notes

`xsd:dateTime` is `CCYY-MM-DDThh:mm:ssZ`