Documentation around the FW4EX schema
Version: 1751

Christian Queinnec

christian.queinnec@paracamplus.com

August 21, 2018

This document 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.rng` (in RelaxNG compact form) rules them all. This http://paracamplus.org/grammar is available from the site of `paracamplus.org`, the open-source side of Paracamplus.

The rest of this chapter is directly generated from `fw4ex.rnc` file.

0.1 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.
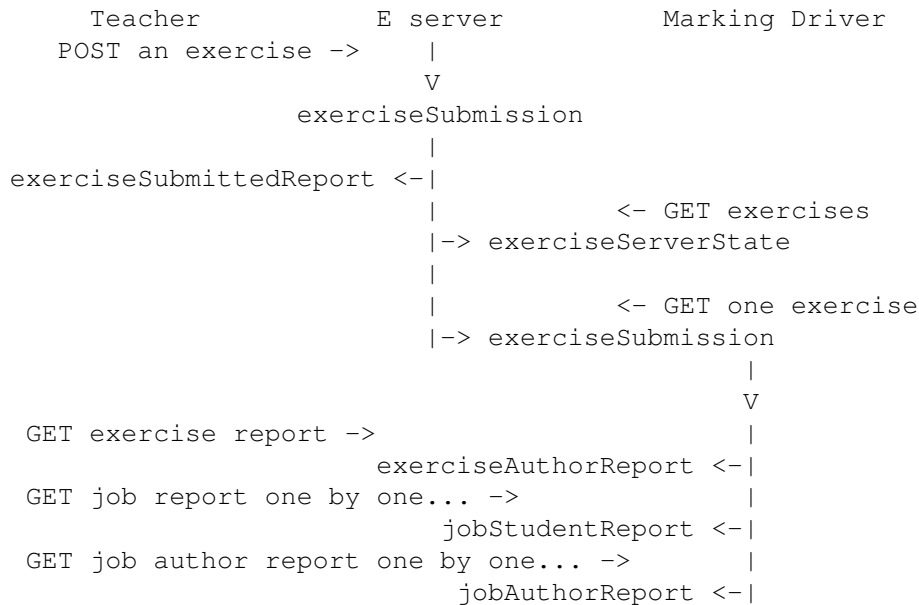
0.2 Use cases

0.2.1 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

4

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.

```
   Student                A server            Marking Driver
 POST a job ->                |
                             V
                     jobSubmission
                            |
   jobSubmittedReport <-|
                            |
                            |              <- GET jobs
                        |-> acquisitionServerState
                            |
                            |              <- GET one job
                        |-> jobSubmission
                                              |
                                              V
 GET job report ->                            |
                     jobStudentReport <-|
```

0.2.2 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.

```
     Teacher              A server            Marking Driver
 POST a batch:                |
  multiJobSubmission ->       |
                             V
                     batchSubmission
                     jobSubmission (many)
                            |
multiJobSubmittedReport <-|
                            |              <- GET batches
                        |-> acquisitionServerState
                            |
                            |              <- GET job one by one...
                        |-> jobSubmission
                            |
                            |              <- GET also the batch
                        |-> batchSubmission
                                              |
                                              V
 GET batch report ->                          |
                     multiJobStudentReport <-|
 GET job report one by one... ->              |
                     jobStudentReport <-|
```

0.2.3 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.........

```
     Teacher              E server           Marking Driver
   POST an exercise ->       |
                             V
               exerciseSubmission
                             |
 exerciseSubmittedReport <-|
                             |              <- GET exercises
                             |-> exerciseServerState
                             |
                             |              <- GET one exercise
                             |-> exerciseSubmission
                                             |
                                             V
 GET exercise report ->                      |
                       exerciseAuthorReport <-|
 GET job report one by one... ->             |
                           jobStudentReport <-|
 GET job author report one by one... ->      |
                            jobAuthorReport <-|
```

0.3 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
  )
}
```

0.4   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
    }
 }
```

0.5 **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
   }
}
```

0.6 **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 }
    } *
}
```

0.7 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
}
```

0.8 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 }
    } *
}
```

0.9 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
   }
}
```

0.10   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
}
```

0.11   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
   } *
 }
```

0.12 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
       } *
```

```
    } ?
 }
```

0.13 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 ?
   } *
}
```

0.14 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 exercise
   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 } ?
}
```

0.15   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 docu-
ment 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
   } ?
}
```

0.16   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
}
```

0.17 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 }
   } ) *
}
```

0.18 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 }
   } *
}
```

0.19 jobsList

This element lists a series of jobs related to an exercise.

```
jobsList = element jobsList {
   exercise.id,
```

14

```
    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 }
        }
    } *
}
```

0.20   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 }
         } *
    }
 }
```

0.21   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 }
   } *
}
```

0.22 groupsReport

This document lists all groups. This requires to be admin.

```
groupsReport = element groupsReport {
  attribute synthetized { xsd:dateTime },
  element group {
    attribute groupName { xsd:NMTOKEN }
  } *
}
```

0.23 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 }
   }
}
```

0.24 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 }
}
```

0.25 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 }
        }
    } +
}
```

0.26 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 }
}
```

0.27 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 } ?
}
```

0.28 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
}
```

0.29 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 ?
}
```

0.29.1   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 }
   }
}
```

0.30   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
}
```

0.31   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
    } ?
}
```

0.32   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 ???
   } ?
}
```

0.33 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 }
    )
```

0.34 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 +
 }
```

0.35 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
}
```

0.36 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
}
```

0.37 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
```

0.38 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
```

0.39 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 }
   } *
}
```

0.40 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 +
}
```

0.41 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 ) +
}
```

0.42 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
```

0.43 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
`command`s 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 +
}
```

0.44 limit

Limits include timeout, cpu, diskio, etc. The name of these limits are predefined (ac-
cording 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.
    } ?
 }
```

0.45   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
    ) +
 }
```

0.46   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 }
   )
}
```

0.47 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
   )
}
```

0.48 command

A command may be predefined or may refer to a script.

0.49 predefined.action

Currently, there is only one predefined action: the `echo` action (reminiscent of the similar task from Ant).

0.50 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
}
```

0.51   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
```

0.52   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 *
```

0.53 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
}
```

0.54 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.

0.54.1 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
```

0.54.2 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
```

0.54.3 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 }
}
```

0.54.4 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
```

0.54.5 component

A command is compound of the name of a program to run and some `component`s specifying with which command line arguments, which input streams, etc.

0.55 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
}
```

0.56 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 }
   )
}
```

0.57 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 *
}
```

0.58 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
   } ?
}
```

0.59 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 ) *
```

0.60 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 ) *
}
```

0.61   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 }
}
```

0.62   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
```

0.63   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 }
 }
```

0.64   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 }
 }
```

0.65   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
```

0.65.1   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
}
```

0.65.2   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
    ) +
}
```

0.66 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
   ) +
}
```

0.67 error

This element is used to notify an error to the student.

```
fw4ex.error = element error {
   (
      xhtml.inline.text
    | xhtml.codeblock
   ) +
}
```

0.68 success

This element is used to notify a success to the student.

```
fw4ex.success = element success {
   (
      xhtml.inline.text
    | xhtml.codeblock
   ) +
}
```

0.69 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
}
```

0.70 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
    ) +
}
```

0.71 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
    ) *
}
```

0.72 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
```

0.73 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 }
}
```

0.74 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
}
```

0.75 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
}
```

0.76 Final notes

`xsd:dateTime` is `CCYY-MM-DDThh:mm:ssZ`