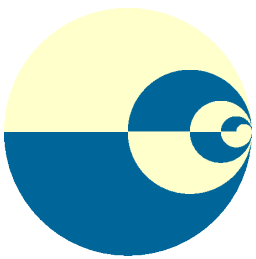
D'un MOOC à l'autre

Performance et généricité dans CodeGradX

C. Queinnec
Pr. émérite UPMC - LIP6



LRDE - 17 décembre 2014

MOOC

- Massive (Small)
- Open
- Online
- Course

MOOC = Enseignement à distance
+ réseau social
+ évaluation continue
+ élasticité

Plan

- Le MOOC « Programmation récursive »
- L'infrastructure CodeGradX
- Particularisation pour le MOOC
- Travaux futurs

Le MOOC « Programmation récursive »

Buts

- Introduction à l'informatique (dans la lignée du SICP) pour débutants, lycéens, enseignants d'ISN
- Fondé sur un cours créé en 2000 et encore donné en 2013 à l'UPMC en L1 S1
  - Livre de cours, livre d'exercices, videos (2004 iTunes), bandes sons (mp3)
  - Nouveaux matériaux en CC-BY-NC-SA
- Doté d'exercices à correction automatisée
- Accent mis sur les tests (*Test Driven Development*)
- Avantages : récursion, Scheme, non-genré

Séquencement

- 10 semaines de cours (mars-mai 2014)
- parution tous les mardis matin (vidéos, transparents, exercices)
- et une brève dans le forum
- inscription non obligatoire (mais sans accès au forum ni aux exercices)
- ressources toujours en ligne (même si MOOC fini)

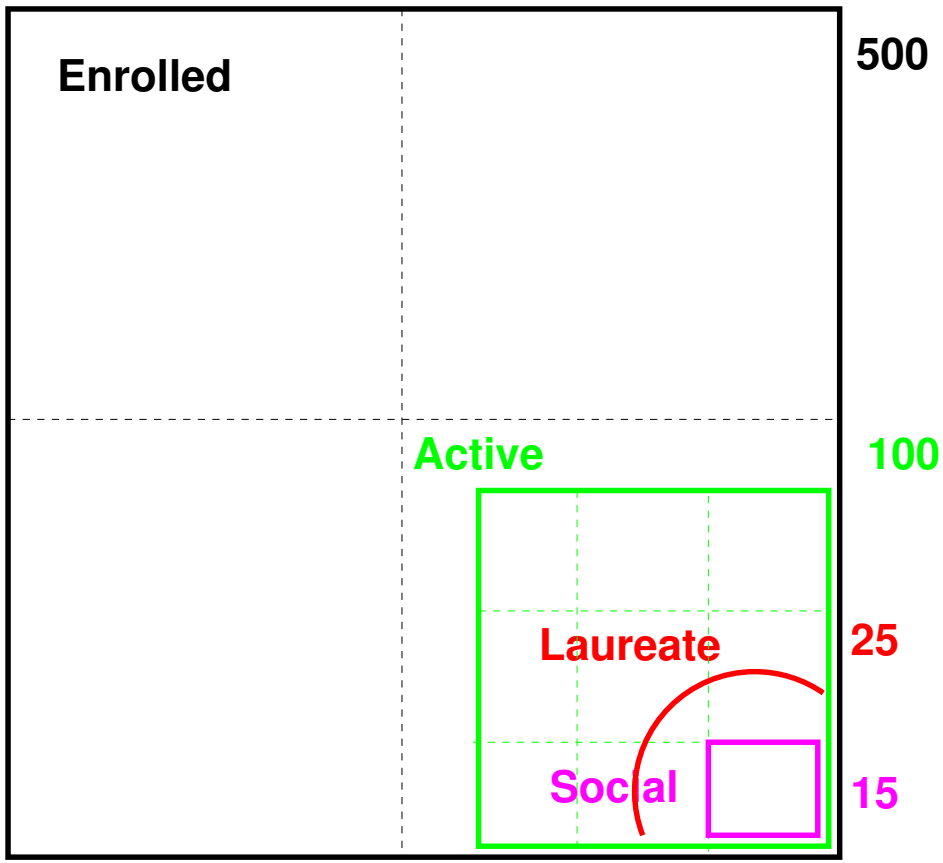
Contenu :

1. Récursion sur entiers naturels
2. Récursion sur listes
3. Récursion sur arbres
4. Processus d'évaluation (texte $\rightarrow$ valeur)

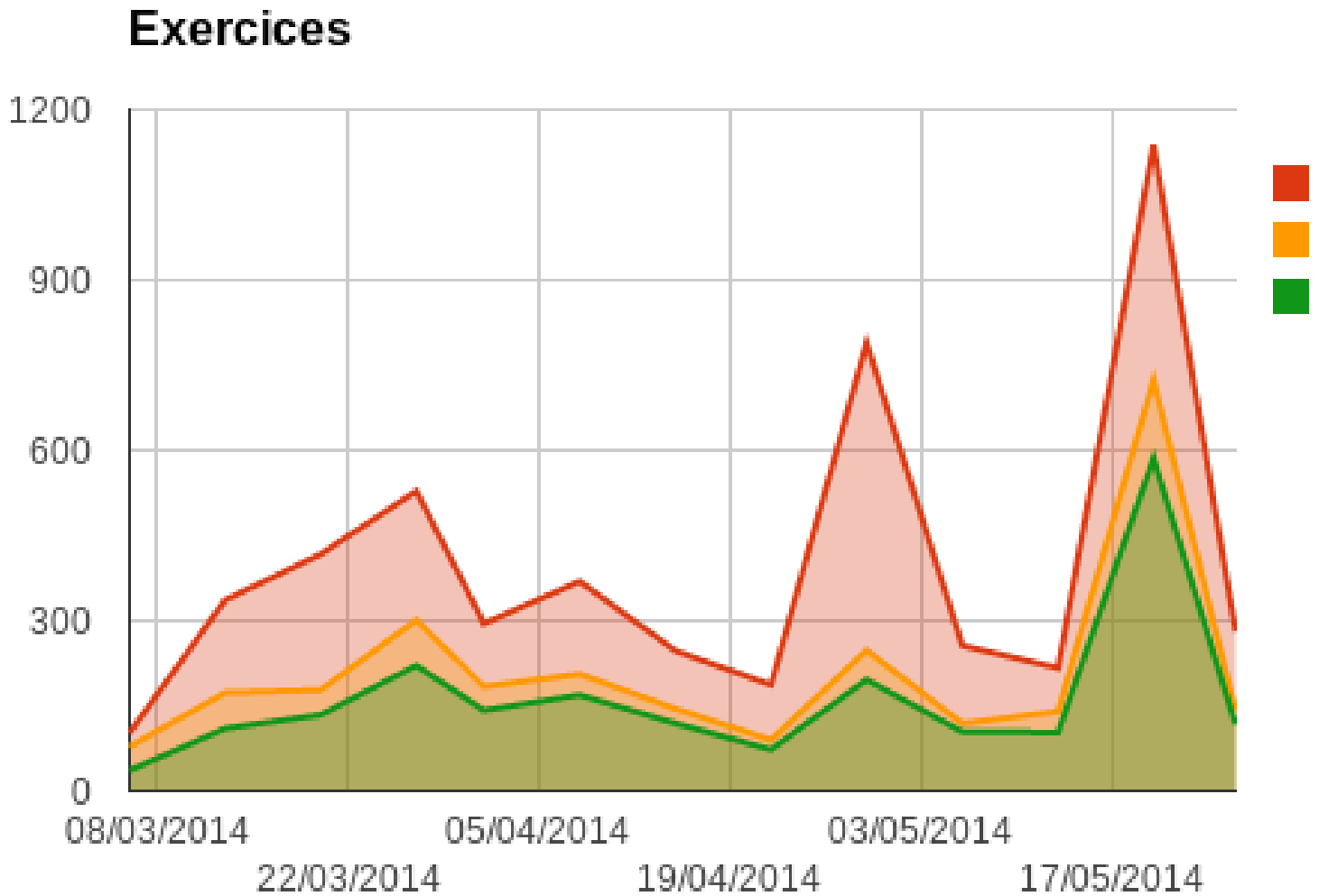
Chiffres finaux (au 27 mai 2014)

inscrits	585
inscrits avec nom-prénom	144
inscrits au forum	110
intervenants au forum	13
sujets, messages sur forum	114, 450
ayant tenté au moins 1 exercice	106
ayant tenté au moins 10 exercices	74
réponses au questionnaire 1	48
réponses au questionnaire 2	11
badgés récursion linéaire	14
badgés récursion arborescente	6
ayant réussi au moins la moitié des exercices à au moins à 80%	26
ayant tenté l'examen	25+10
certificats attribués	24

Résumé des chiffres finaux (au 27 mai 2014)

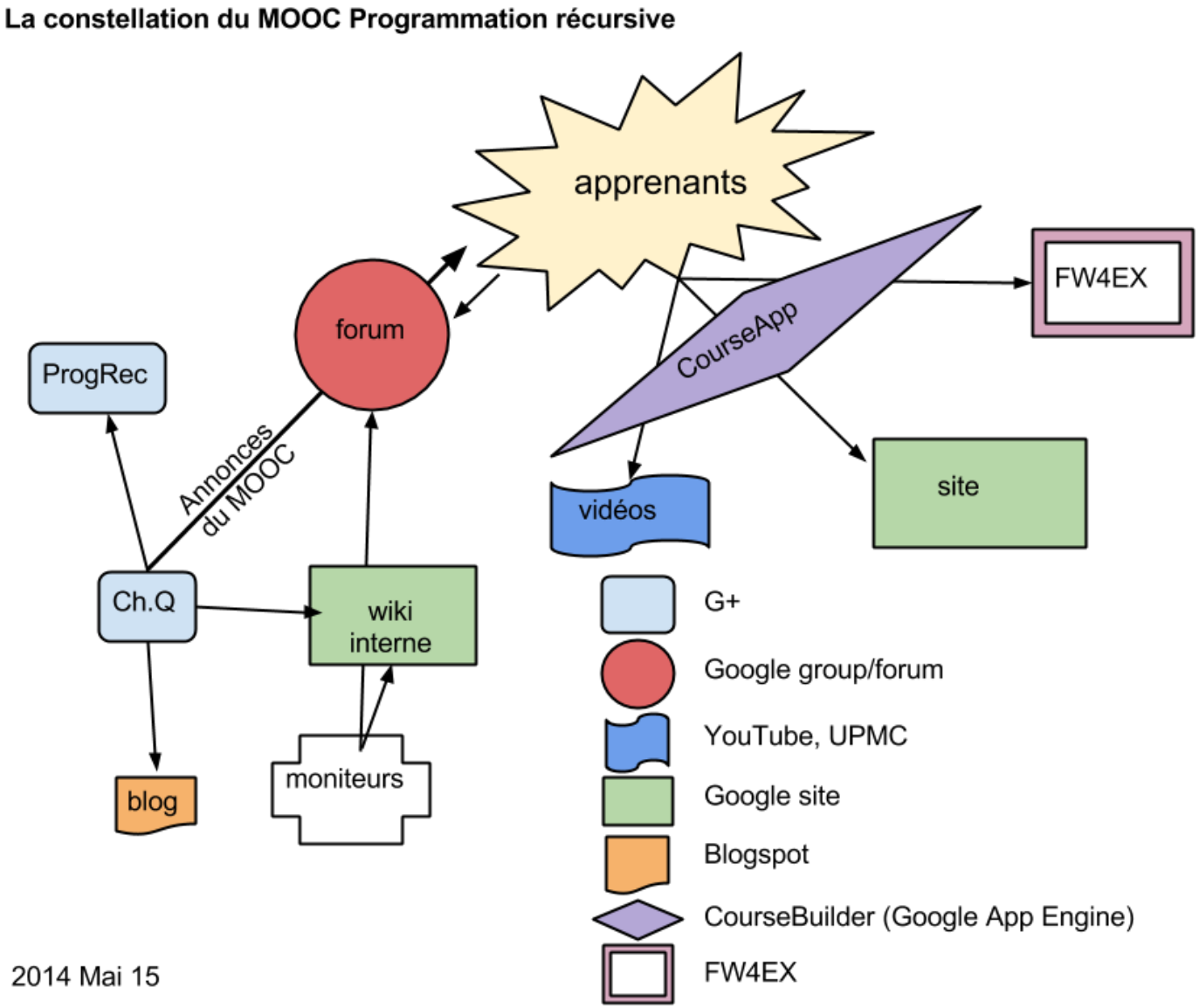


Exercices / semaine



nombre d'essais, réussites à 80% ou plus, à 100%.

De bric et de broc (merci Google)



Infrastructure CodeGradX

À la base

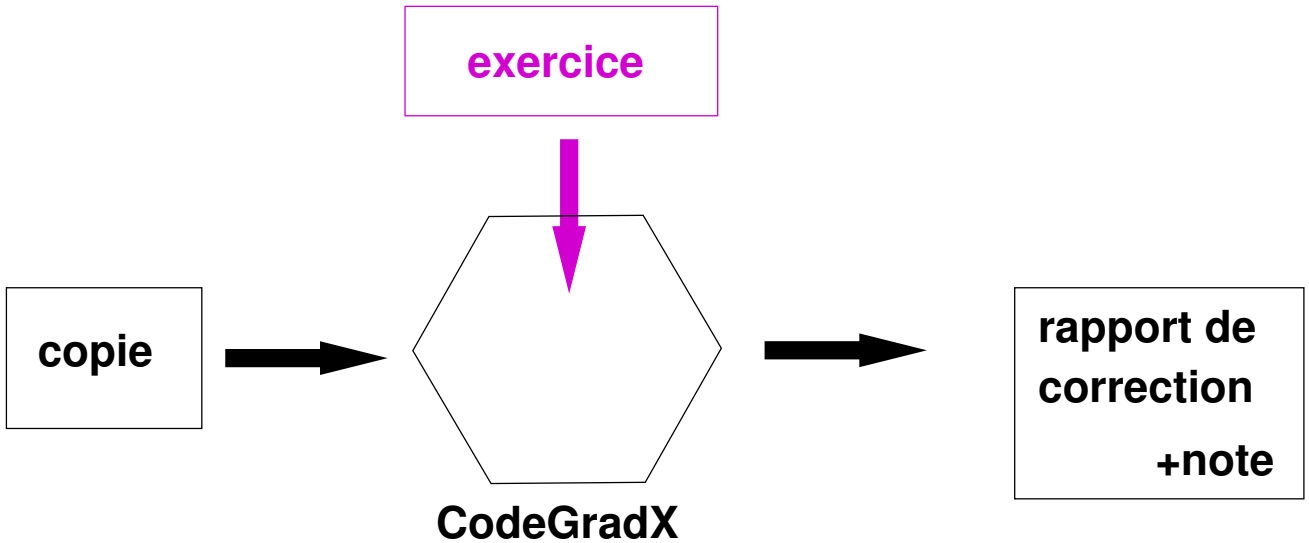
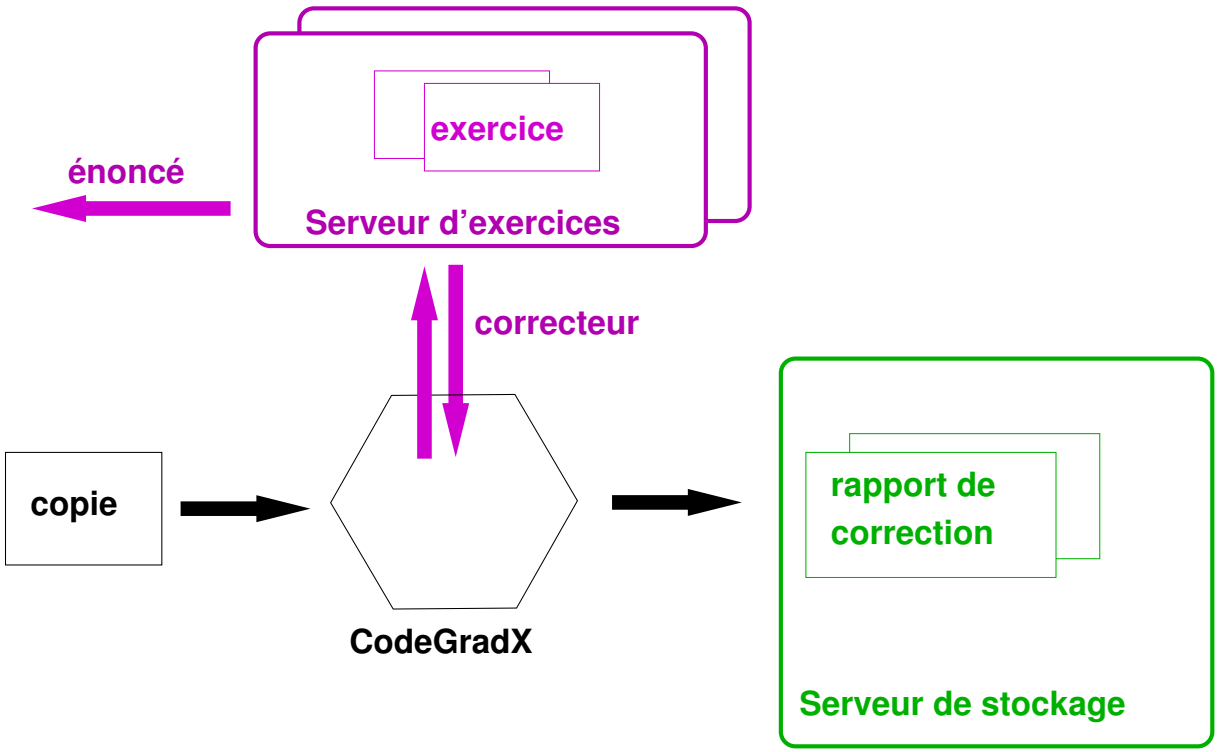
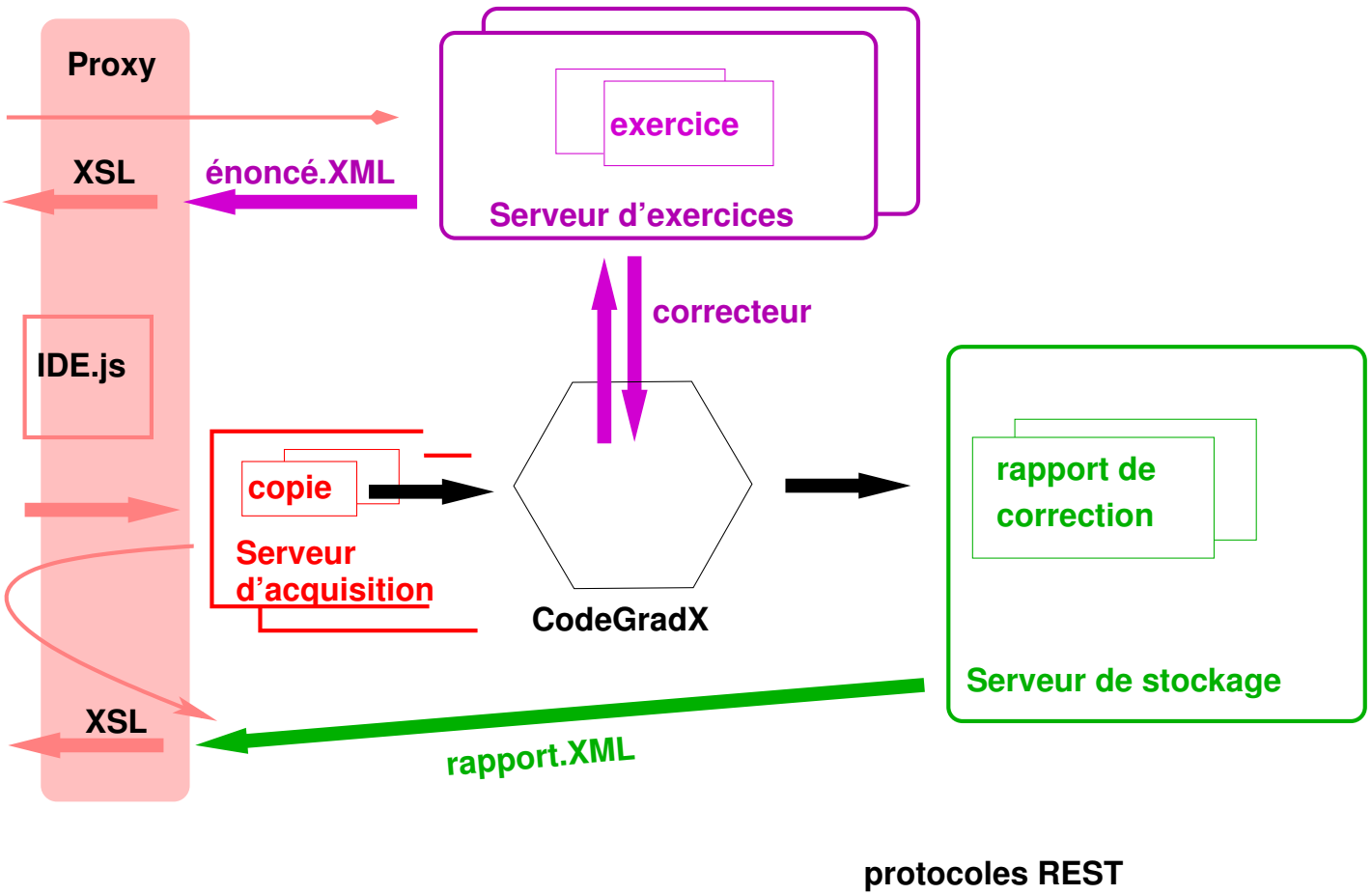


Plate-forme en fonctionnement depuis 2008 et 140 000 copies corrigées

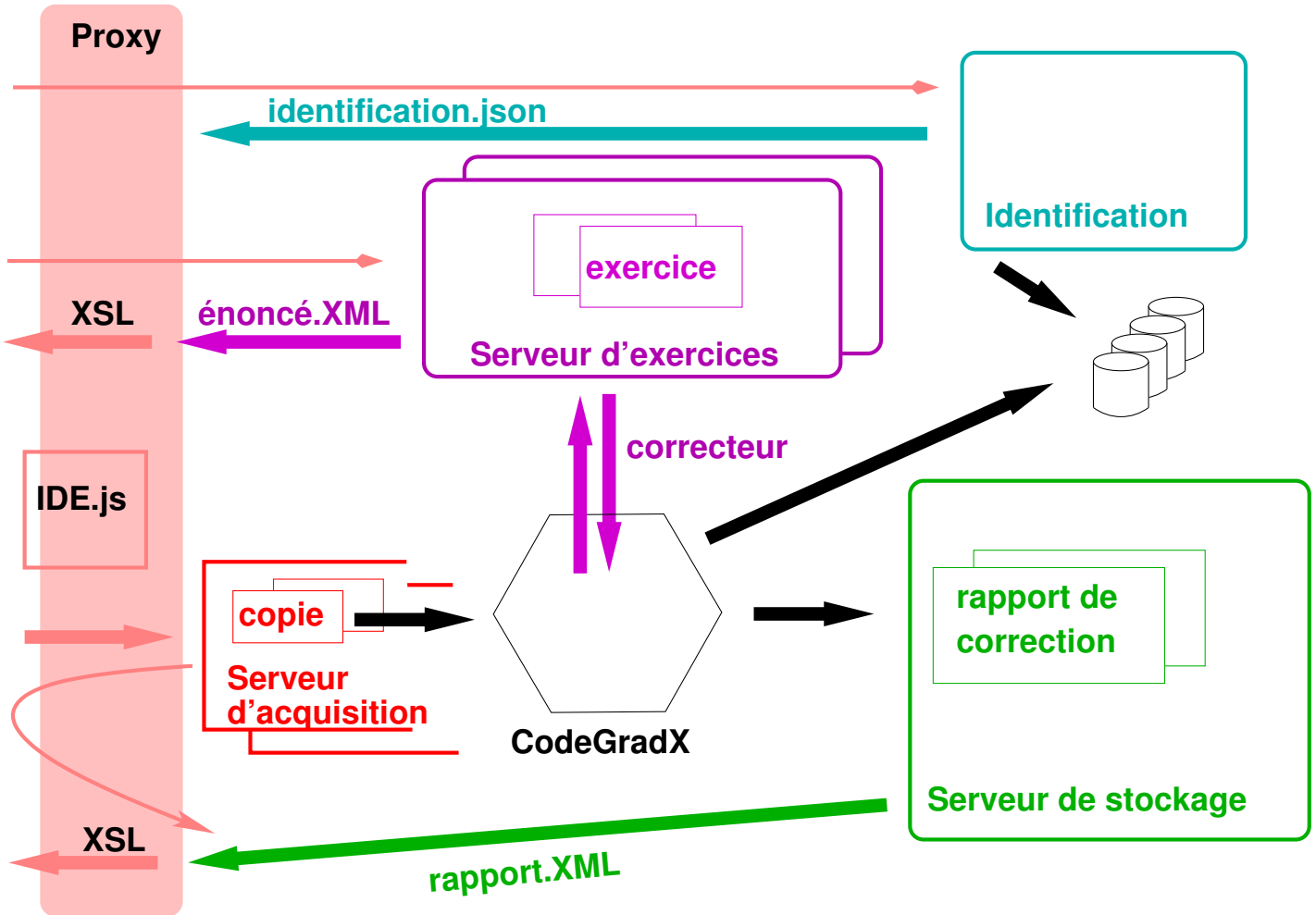
Serveurs spécialisés



Particularisation via proxy



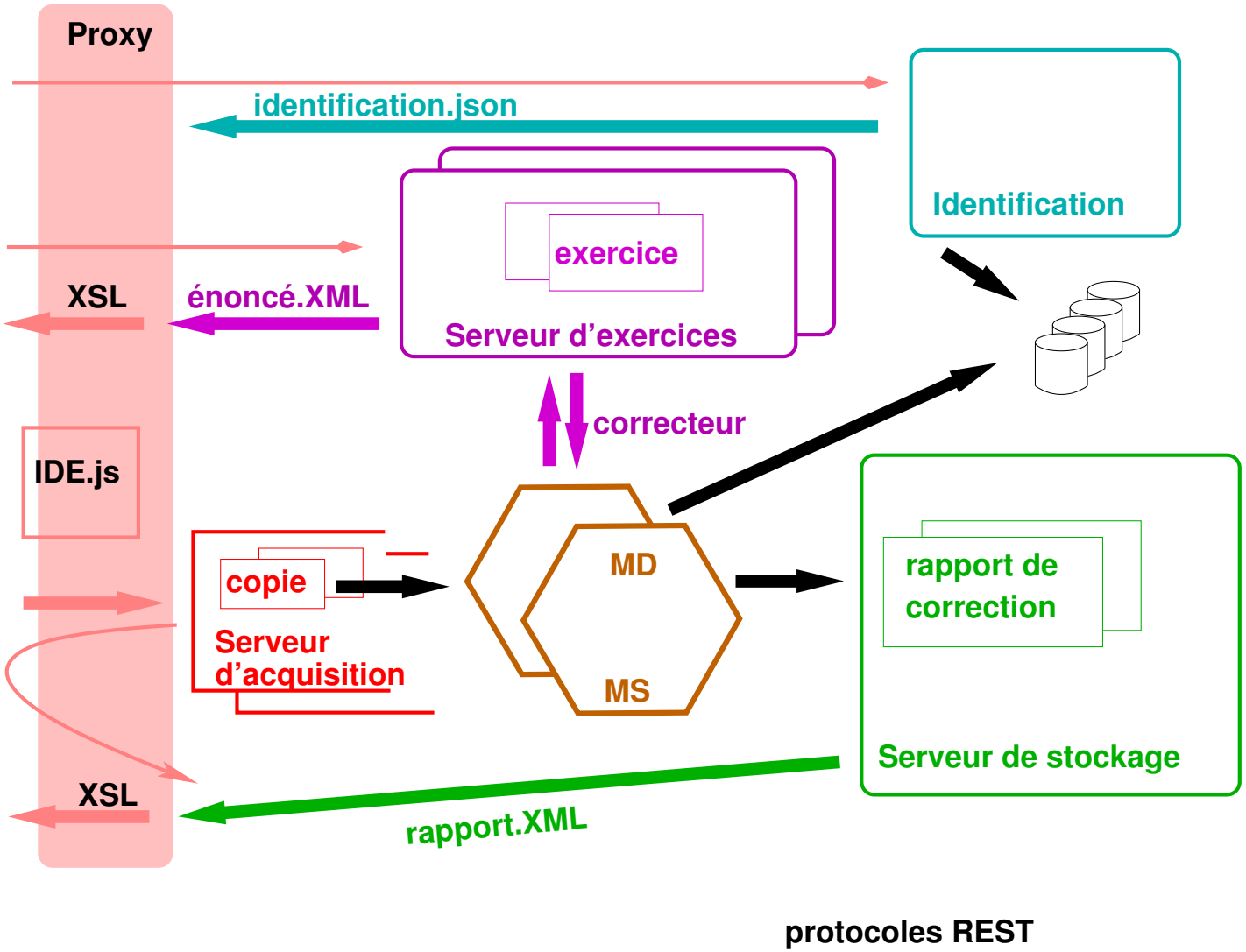
Identification et bases de données



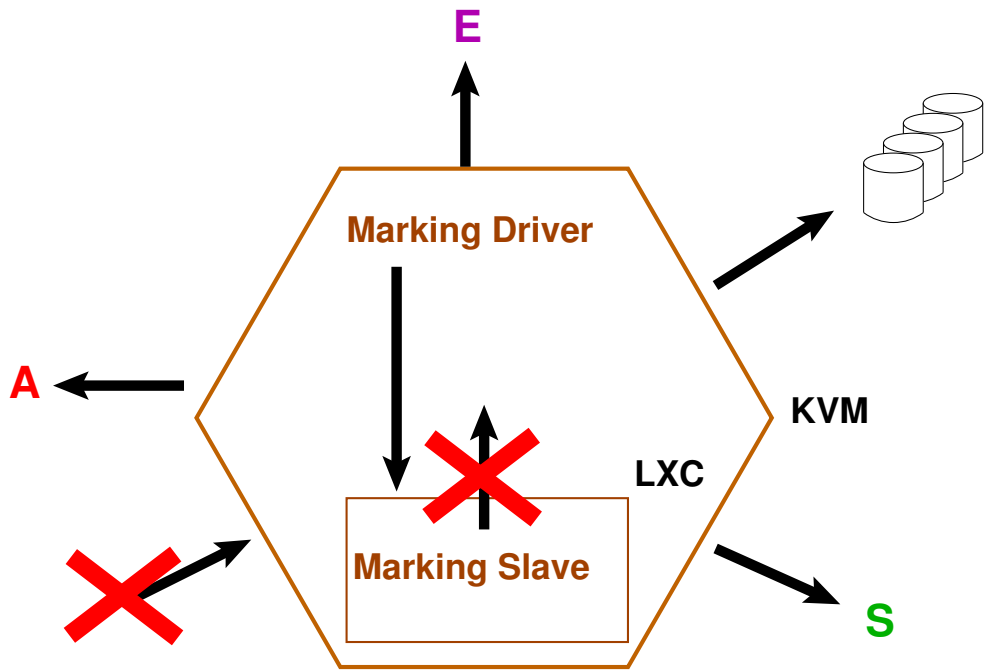
Safecookies

- Les serveurs d'acquisition et d'exercices ne nécessitent pas d'accès à la base mais n'acceptent de requêtes que d'utilisateurs autorisés.
- Une identification réussie mène à un cookie contenant une sorte de certificat que peuvent vérifier ces serveurs.
- Les exercices sont cherchés avec une URL contenant aussi une telle sorte de certificat afin d'éviter d'être pillé.
- Les rapports sont protégés par des UUID.

Multiples correcteurs en parallèle



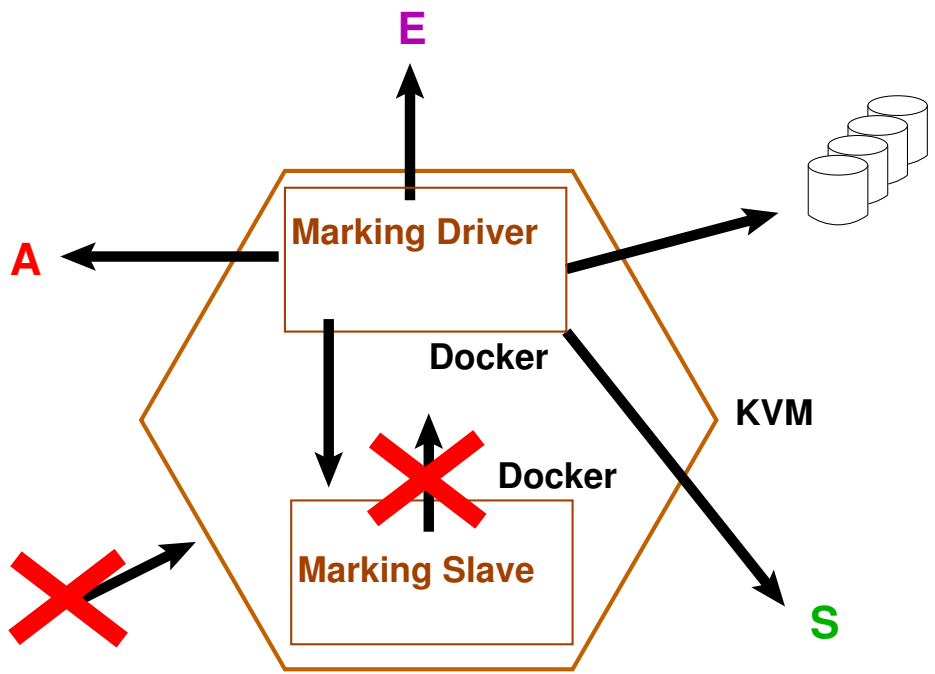
Correcteur confiné



- VM (kvm, libvirt, vmware, virtualbox)
- disque de 12G
- assure le gel des bibliothèques.
- Seul MD connaît les identités des apprenants.

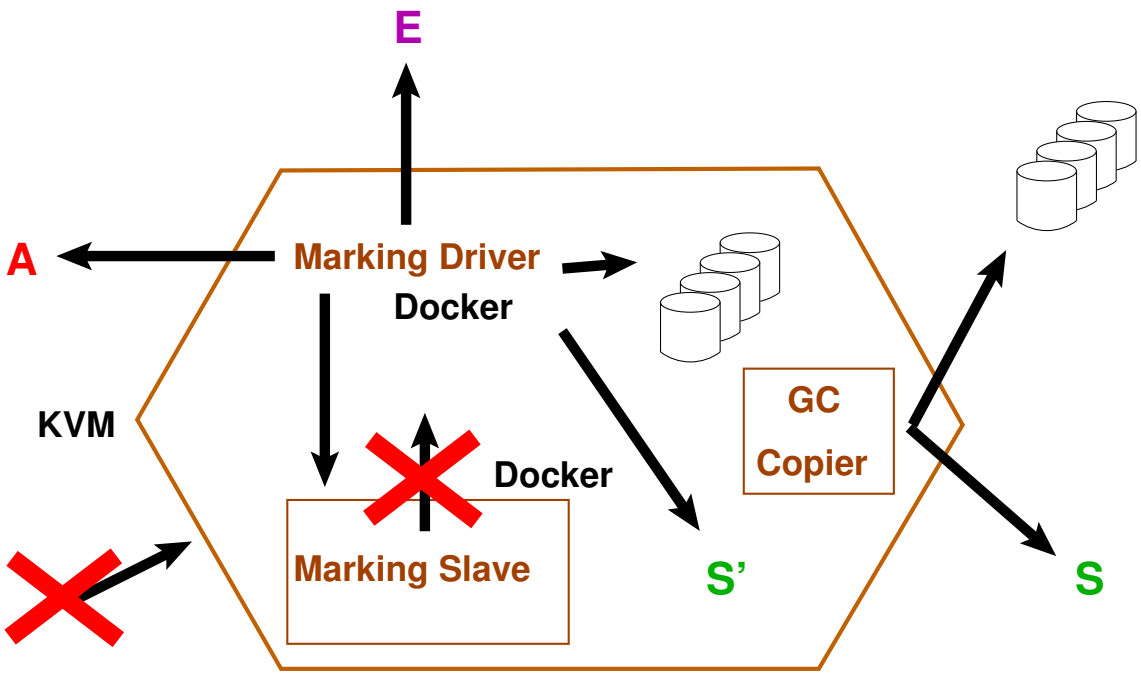
Version actuelle des correcteurs confinés

Une VM encapsulant deux images Docker :



- règle le problème de la mise à jour

Déconnexion réticulaire



Exercice

Un exercice est juste un tar.gz, à déploiement aisé, contenant :

- un énoncé
- des scripts de correction
- des pseudo-copies
- et un descripteur XML

Les scripts sont confinés en fonctionnalités, en temps et en production d'octets.

Actuellement existent des exercices pour C, Java, bash, Octave, Scheme, OCaml, Python.

L'éternité pour vos exercices

Scripts de correction

Code appartenant à l'auteur de l'exercice mais exécuté dans le HOME de l'étudiant.

- exercice = ensemble de questions
- question = suite de scripts
  - par script :
    - directives de confinement
    - code du script
    - stdout vers l'apprenant (y compris notes partielles)
    - stderr vers l'auteur de l'exercice
  - stdout vers auteur
  - stderr vers mainteneur CodeGradX
- Compilation de l'enchaînement vers Bash
- Nettoyage du flux XML produit vers l'apprenant

Cycle de vie d'un exercice

	MD	**MS**	**e**
job→			
	ex.tgz←		ex.tgz
	ex.tgz→	installe	
	job→	corrige	
	rapport←	rmStudent	
rapport	←tidy		
		...	
		plus tard	
		desinstalle	

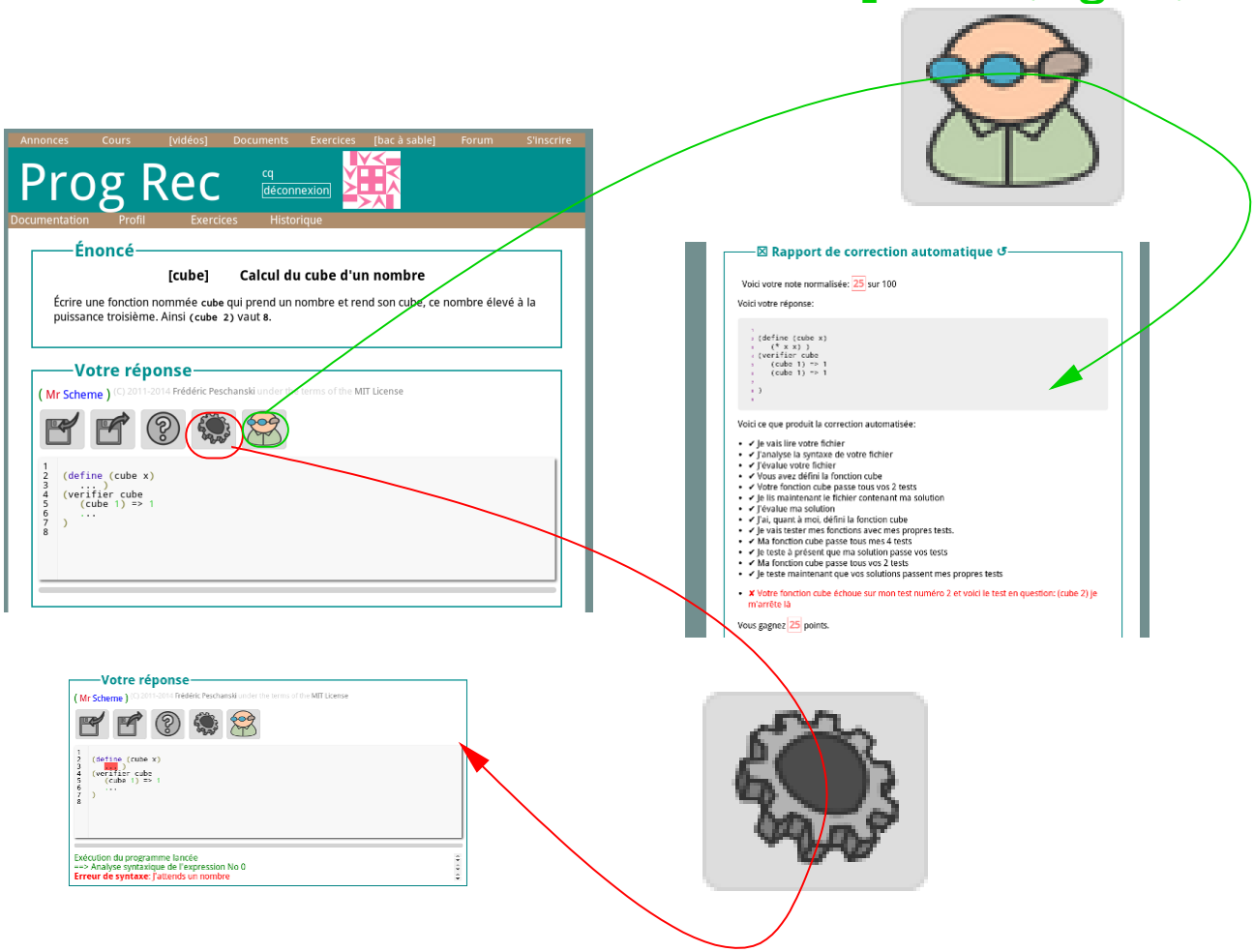
Conseils pour exercices

- Écrire énoncé, solution et correcteur en même temps
- Verbaliser la correction
- Pas de question tout ou rien
- Ne pas oublier de confiner
- Écrire des pseudo-copies
- . . . les enrichir au gré des expériences
- Ne pas perturber le HOME de l'étudiant
- Utiliser TMPDIR
- Factoriser code commun dans l'installation de l'exercice
- Utiliser bibliothèque tests unitaires progressifs
    - Scheme, Java, Python

Cas du MOOC

Environnement de développement

Distant Scheme interpreter (bigloo)



Local Scheme Interpreter (Javascript)

Notation

Le langage étant fonctionnel, on demande des fonctions.
L'apprenant écrit (et l'auteur de l'exercice itou) :

```
(define (foo ...)                    f_s
   ... )
(verifier foo                        v_s
  (foo ...) => ...      ; au moins 2 tests
  (foo ...) => ... )
```

1. cohérence(0% éliminatoire) : on vérifie $v_s(f_s)$
2. correction1(0% (Ouf !)) : on vérifie $v_s(f_t)$
3. correction2(50%) : on vérifie $v_t(f_s)$
4. couverture(50%) : on compare $v_s(f_s)$ et $v_t(f_s)$

De l'importance de la verbalisation

☒ Rapport de correction automatique ↺

Voici votre note normalisée: 25 sur 100

Voici votre réponse:

```
1
2 (define (cube x)
3    (* x x) )
4 (verifier cube
5    (cube 1) => 1
6    (cube 1) => 1
7
8 )
9
```

Voici ce que produit la correction automatisée:

- ✔ Je vais lire votre fichier
- ✔ J'analyse la syntaxe de votre fichier
- ✔ J'évalue votre fichier
- ✔ Vous avez défini la fonction cube
- ✔ Votre fonction cube passe tous vos 2 tests
- ✔ Je lis maintenant le fichier contenant ma solution
- ✔ J'évalue ma solution
- ✔ J'ai, quant à moi, défini la fonction cube
- ✔ Je vais tester mes fonctions avec mes propres tests.
- ✔ Ma fonction cube passe tous mes 4 tests
- ✔ Je teste à présent que ma solution passe vos tests
- ✔ Ma fonction cube passe tous vos 2 tests
- ✔ Je teste maintenant que vos solutions passent mes propres tests

- ✘ Votre fonction cube échoue sur mon test numéro 2 et voici le test en question: (cube 2) je m'arrête là

Vous gagnez 25 points.

Un brin de sémantique

```
;; code etudiant
(define (bar t u) ...)
(define (foo x)
  (bar x x) )
(verifier foo
  (foo 1) => 2 )
```

`foo` utilise `bar`

Un brin de sémantique (2)

```
;; code etudiant
(define (bar t u) ...)
(define (foo x)
  (bar x x) )
(define (hux z) ...)
(verifier foo
  (hux (foo 1)) => 2 )
```

mais vérifier `foo` nécessite `hux`

Un brin de sémantique (3)

```
;; code etudiant
(define (bar t u) ...)
(define (foo x)
  (bar x x) )
(define (hux z) ...)
(verifier foo
  (hux (foo 1)) => 2 )
```

```
;;; Code enseignant
(define (foobar x) ...)
(define (foo t)
   (foobar t) )

(verifier foo
  (foo 2) => 3 )
```

Mais $\texttt{foo}_t$ nécessite $\texttt{foobar}_t$
Mais tester $\texttt{foo}_t$ utilise $\texttt{hux}_s$.

Un brin de sémantique (4)

```
;; code etudiant
(define (bar t u) ...)
(define (foo x)
  (bar x x) )
(define (hux z) ...)

(verifier foo
  (hux (foo 1)) => 2 )
```

```
;;; Code enseignant
(define (foobar x) ...)
(define (foo t)
   (foobar t) )
(define (boolify a)
  (and a #t) )
(verifier (foo)
  (boolify (foo 2))
  => 3 )
```

Les codes sont clos dans leur environnement global de définition.

Sur-implantation

Le code de l'apprenant est comparé au code de l'enseignant. Malheureusement, le code de l'enseignant doit être plus précautionneux que demandé par l'énoncé.

- S'il est demandé une fonction `foo: nat -> X`
- et que l'apprenant code `foo: int -> X`
- et qu'il teste `(foo -1)` alors
- il est hors épure et $v_s(foo_t)$ échouera !
- Pour donner un message signifiant, $\texttt{foo}_t$ doit tester explicitement que l'argument est bien un `nat`.

Travaux futurs

En cours

- *ε-better peeping*
- Sélection de copies aidantes ou intéressantes
- Annotation de copies
- Duo-duel
  - éditeur partagé
- Proposition d'exercices

Futur

MOOC cherche parrainage

Des questions ?