Recherche reproductible avec Lepton

Li-Thiao-Té Sébastien

LAGA UMR 7539, Université Paris 13

Outline

Préliminaire

Code chunk 1: ≪shell≫

```
lepton -env verbatim lepton_baretex.nw
```

Est-ce que ça marche ?

- Sauvegarder le fichier `rapport.nw` dans un répertoire
- Lancer la commande

Code chunk 2: ≪`shell (part 2)`≫

```
lepton.first -env verbatim rapport.nw
```

Interpret with `shell`

```
apprentissage.txt (part 1 write): chunk as text,
code.R (part 1 write): chunk as s, exec with R, output as text,
code.R (part 2 write): chunk as s, exec with R, output as text,
data.txt (part 1 write): chunk as text,
code.R (part 3 write): chunk as s, exec with R, output as text,
shell (part 1): chunk as sh, exec with shell, output as text,
```

Est-ce que ça marche ?

- Sauvegarder le fichier `rapport.nw` dans un répertoire
- Lancer la commande
- Vérifiez que vous avez bien les fichiers suivants dans le répertoire :

Code chunk 6: «`shell (part 3)`»

```
ls -al data.txt code.R result.txt rapport.nw rapport.tex
```

Interpret with `shell`

```
-rw-r--r-- 1 lithiao lithiao  948 Mar 28 09:02 code.R
-rw-r--r-- 1 lithiao lithiao   36 Mar 28 09:02 data.txt
-rw-r--r-- 1 lithiao lithiao 3092 Mar 28 09:02 rapport.nw
-rw-r--r-- 1 lithiao lithiao 4425 Mar 28 09:02 rapport.tex
-rw-r--r-- 1 lithiao lithiao   91 Mar 28 09:02 result.txt
```

Est-ce que ça marche ?

- Sauvegarder le fichier `rapport.nw` dans un répertoire
- Lancer la commande
- Vérifiez que vous avez bien les fichiers suivants dans le répertoire :
- Compiler le fichier `rapport.tex`.

Code chunk 10: ≪`shell (part 4)`≫

```
pdflatex --interaction=batchmode rapport.tex
```

Interpret with `shell`

```
This is pdfTeX, Version 3.1415926-2.4-1.40.13 (TeX Live 2012/Debian)
 restricted \write18 enabled.
entering extended mode
```

Félicitations

Vous avez réussi à :

- récupérer mon rapport de projet `rapport.nw`.
- récupérer les données du problème `data.txt`.
- récupérer le code source `code.R`.
- récupérer les commandes et leurs paramètres
- générer les résultats `result.txt`.
- générer les figures et le PDF `rapport.pdf`.
- réutiliser mon travail (ou presque).

Lepton

- OCaml
- Site web : `http://www.math.univ-paris13.fr/~lithiao/ResearchLepton/Lepton.html`
- avec le programme en libre téléchargement pour Linux 32/64, (Windows)
- manuel + faq + exemples
- 2 conference papers
  - Sébastien Li-Thiao-Té. Literate program execution for reproducible research and executable papers. *Procedia Computer Science*, 9(0) :439 – 448, 2012. ICCS 2012.
  - Sébastien Li-Thiao-Té. Literate program execution for teaching computational science. *Procedia Computer Science*, 9(0) :1723 – 1732, 2012. ICCS 2012.

À quoi ça sert ?

Écrire des documents au sens large

- Travaux de recherche
  - articles de journaux
  - présentations beamer
  - rapports techniques
- Enseignement
  - diapositives de cours
  - exercices aléatoires
  - sujets d'examen aléatoires
  - avec leur solution !
- Autres
  - analyser les résultats d'une enquête
  - analyser la réussite des élèves selon leur parcours scolaire
  - élaboration d'un budget
  - site web

Outline

Fonctionnalités

Lepton est un programme qui traite des blocs de texte

Code chunk 11: ≪lepton≫

```
<<name options>>=
code
@
```

avec quatre opérations de base :

- `-write` permet d'écrire sur le disque
- `-exec interpreter` permet d'exécuter le code
- `-chunk format -output format` contrôle le format de sortie
- `<<chunk_ref>>` permet de réutiliser un autre bloc

Écrire sur le disque

Code chunk 12: ≪ecrire≫

```
<<fichier.txt -write>>=
contenu du fichier
@
```

Code chunk 13: ≪shell (part 5)≫

```
cat fichier.txt
```

Interpret with `shell`

```
contenu du fichier
```

Exécution de commandes

Code chunk 14: ≪ecrire (part 2)≫

```
<<shell -exec shell>>=
uname -a
@
```

Code chunk 15: ≪shell (part 6)≫

```
date
uname -a
```

Interpret with `shell`

```
Thu Mar 28 09:02:32 CET 2013
Linux laptop 3.2.0-4-686-pae #1 SMP Debian 3.2.35-2 i686 GNU/Linux
```

Format

Code chunk 16: ≪ecrire (part 3)≫

```
<<caml -exec ocaml -chunk ocaml>>=
let rec split = function
  | a :: b :: c -> let l1,l2 = split c in (a :: l1), (b :: l2)
  | a :: [] -> [a],[]
  | [] -> [],[]
;;
@
```

Code chunk 17: ≪caml≫

```
let rec split = function
  | a :: b :: c -> let l1,l2 = split c in (a :: l1), (b :: l2)
  | a :: [] -> [a],[]
  | [] -> [],[]
;;
split [1;2;3;4;5];;
```

Interpret with ocaml

```
val split : 'a list -> 'a list * 'a list = <fun>
- : int list * int list = ([1; 3; 5], [2; 4])
```

Format

Code chunk 18: ≪ecrire (part 4)≫

```
<<R -exec R -chunk hide -output verb>>=
data = round(runif(5),2)
cat("\\frac{",paste(data,collapse="+"),"}{",length(data),"}\n")
@
```

$$\bar{x} = \frac{0.44 + 0.61 + 0.18 + 0.28 + 0.76}{5}$$

Inclusion directe dans la documentation.
Par exemple, \Lexpr{R}{cat(data)} donne 0.44 0.61 0.18 0.28 0.76.

Réutilisation de commandes

The example program computes the value of the Bessel function $J_0(5)$:

$$J_\alpha(x) = \sum_{m=0}^{\infty} \frac{(-1)^m}{m!\Gamma(m+\alpha+1)} \left(\frac{x}{2}\right)^{2m+\alpha}$$

Code chunk 19: ≪example.c≫

includes

```
<<includes>>
int
main (void)
{
    double x = 5.0;
    double y = gsl_sf_bessel_J0 (x);
    printf ("J0(%g) = %.18e\n", x, y);
    return 0;
}
```

The include directives are separated for convenience.

Code chunk 20: ≪includes≫

```
#include <stdio.h>
#include <gsl/gsl_sf_bessel.h>
```

Outline

Les quatre R de la recherche (reproductible)

- Re-view : détails de la méthode
- Re-ceive : documentation et pédagogie
- Re-play : reproduction des résultats par un tiers
- Re-use : accès aux différents éléments

Re-view : les détails de la méthode

Inclusion automatique des détails :

- codes sources et scripts
- enchaînement des commandes
- valeurs des paramètres utilisés

Exemples :

- dépendances d'un programme (liste des paquets avec dpkg)

Code chunk 21: ≪shell (part 7)≫

```
COLUMNS=80 dpkg -l gcc ocaml r-base
```

Interpret with shell

```
Desired=Unknown/Install/Remove/Purge/Hold
| Status=Not/Inst/Conf-files/Unpacked/halF-conf/Half-inst/trig-aWait/Trig-pend
|/ Err?=(none)/Reinst-required (Status,Err: uppercase=bad)
||/ Name           Version      Architecture Description
+++-==============-============-============-=================================
ii  gcc            4:4.7.2-1    i386         GNU C compiler
ii  ocaml          3.12.1-4     i386         ML language implementation with a
ii  r-base         2.15.1-4     all          GNU R statistical computation and
```

Re-view : les détails de la méthode

Inclusion automatique des détails :

- codes sources et scripts
- enchaînement des commandes
- valeurs des paramètres utilisés

Exemples :

- dépendances d'un programme (liste des paquets avec dpkg)
- compilation d'un code source (makefile)
- tests

Code chunk 24: ≪shell (part 8)≫

```
gcc -Wall -I/usr/local/include -lgsl -lgslcblas -lm example.c -o a.out 2>&1
./a.out
```

Interpret with shell

```
J0(5) = -1.77596771314338292e-01
```

Re-ceive : documentation

Documentation dans le format de votre choix (LaTeX, HTML, MarkDown, etc.)

- fonctionnalités du format : coloration syntaxique, liens hypertextes, figures, diagrammes, animations
- structure du code
- documentation croisée : algorithme et code source au même endroit
- documentation générée par un programme

Exemples :

- rapport technique
- implémentation détaillée d'un code source
- manuscrits de recherche

Réutilisation de commandes

The example program computes the value of the Bessel function $J_0(5)$:

$$J_\alpha(x) = \sum_{m=0}^{\infty} \frac{(-1)^m}{m!\Gamma(m+\alpha+1)} \left(\frac{x}{2}\right)^{2m+\alpha}$$

Code chunk 25: ≪example.c≫

includes

```
<<includes>>
int
main (void)
{
    double x = 5.0;
    double y = gsl_sf_bessel_J0 (x);
    printf ("J0(%g) = %.18e\n", x, y);
    return 0;
}
```

The include directives are separated for convenience.

Code chunk 26: ≪includes≫

```
#include <stdio.h>
#include <gsl/gsl_sf_bessel.h>
```

Re-play : documents exécutables

Exécution automatique des commandes du document

- extraction / écriture des fichiers
- exécution des commandes et scripts (semi concurrente)
- récupération des résultats

Exemples :

- compilation d'un code source
- démonstration d'un algorithme
- benchmarks
- mise à jour de résultats avec nouvelles données

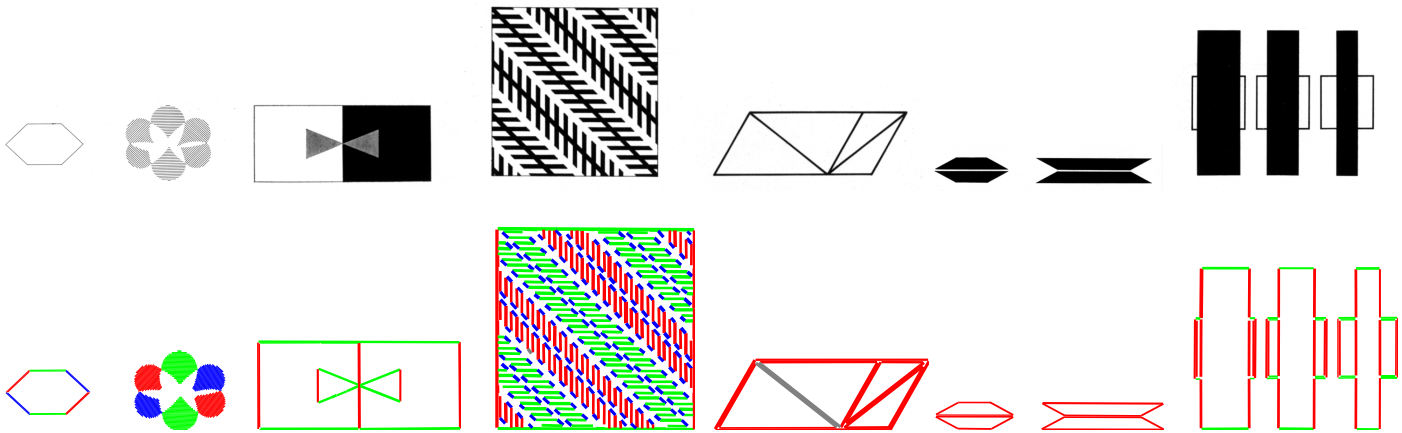

FIGURE: Quelques images et le résultat de leur traitement

Re-use : accès aux éléments

Un bloc de texte peut contenir tout type de texte :

- données d'entrées
- code source et scripts
- résultats

En particulier, ces éléments sont

- extraits automatiquement
- présentés en clair
- documentés

Code chunk 27: ≪xml_example≫

```
  <scan num="48" msLevel="2" peaksCount="22"
        polarity="+" scanType="Full"
        retentionTime="PT18.5663S" collisionEnergy="0"
        lowMz="170" highMz="2000"
basePeakMz="546.767" basePeakIntensity="23.3507" totIonCurrent="161.317">
    <precursorMz precursorIntensity="35143.6">670.4302368</precursorMz>
    <peaks precision="32" byteOrder="network"
    pairOrder="m/z-int">Q3sjJ0CXKNBDq2y8QS9R5kPJ2GxAsuSeQ866tkALVvpDzz7hQH8L1EPTsYZ
  </scan>
```

Conclusion

Lepton est un logiciel qui permet

- d'écrire des documents
- en manipulant la position de blocs de texte (literate programming)
- et leur interprétation (exécution, écriture sur disque)

Qu'est-ce que cela apporte :

- reproductibilité du travail (de recherche)
- documentation croisée de code source, scripts, données
- génération automatique de certaines parties de la documentation
- garanties de cohérence

Questions ?