Scripting languages work methodology

Tomasz Bold
tomasz.bold@fis.agh.edu.pl
D11 – pok. 107

Organisation

- Lectures & labs one after another
  - **Lecure 1 – Elementary information: history, applications, methodology.**
  - **Lecture 2 – bash and co., text processing**
  - labs x 2
  - projects presentation - with discussion
  - **Lecture 3 – python**
  - labe x 3
  - projects presentation
  - **Lecture 4 – ROOT/CINT**
  - labs x 2
  - **Lecture 5 – JavaScript**
  - labs x 2
  - projects presentation

- 3 Projects will need to be prepared by everyone

Final grade

- 30% lab 70% projects
- at the start of each lab a short written test (10 minutes at most)
- 3 - projects completed
  - this projects can be (very welcome) used at other courses
  - however not on dedicated courses (i.e.: python project can not serve here and in dedicated python course)

Let’s start

Today

- Why scripting languages
  - When not to go for scripting
- Proliferation, history, examples
- Programming patter for scripts

What is a script?

"A script is something you give the actors. A program is what you give the audience." – Larry Wall

- No compilation stage
- Aiming at productivity rather than easthetics, simple syntax (TCL syntax: 11 paragraphs, ~2 pages)
- Automatic memory management, very functional build in types (e.g.: rich strings, associative tables)
- Often lack of strict type system (e.g.: all is a string)

Scripts as integration languages

- Not useful for creation of large & complex programs,
in contrary they are a “**glue**” to join programs in a larger systems→ therefore often called system integration languages (e.g.: TCL, Julia)

- Extension/tailoring compiled program functionalities

- Scripted programs performance often depends on integrated within them compiled programs

Justification for writing scripts

A programmer efficiency: M/N
N number of code lines → M number of CPU instructions executed

- Simplest measure of effort is LOC (Line Of Code)
- LOC → programming language → number of instructions —> amount of processing done

N/M example

- One line in bash ~300 CPU instructions
- Typical python program 3 to 10 times shorter as compared to java
- Time (Your time!) correspondingly shorter

Example form the literature

- Lutz Prechelt An empirical comparison of seven programming languages. IEEE Computer 33(10): 23-29, October 2000.
  - Program: phone number converter → word: 048-374363712 == 048-friends-12
    - 3 non-scripting languages (C, C++, Java)
    - 4 scripting (Perl,python, Rexx, TCL),
    - students and other usenet

Results of empirical measurements

- Time spent writing the program & number of lines:

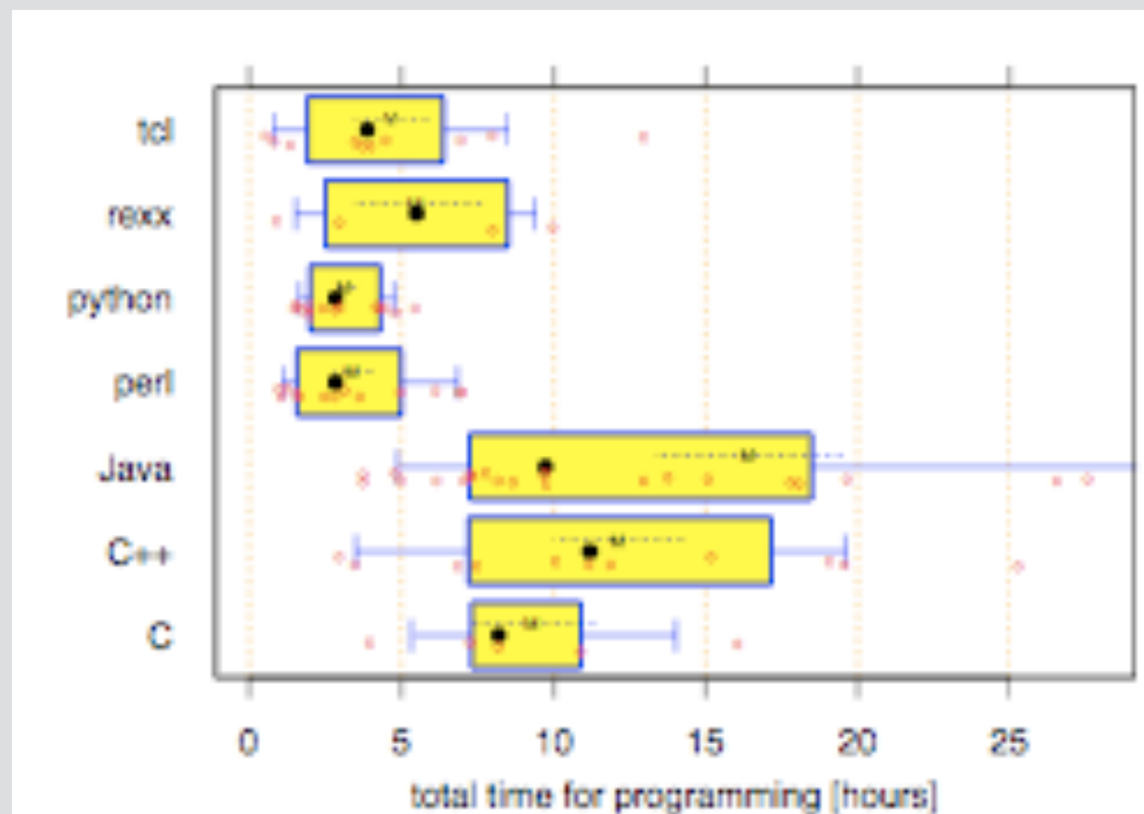


Figure 6: Total working time for realizing the program. Script group: times as measured and reported by the programmers. Non-script group: times as measured by the experimenter. The bad/good ratios range from 1.5 for C up to 3.2 for Perl. Three Java work times at 40, 49, and 63 hours are not shown. ••

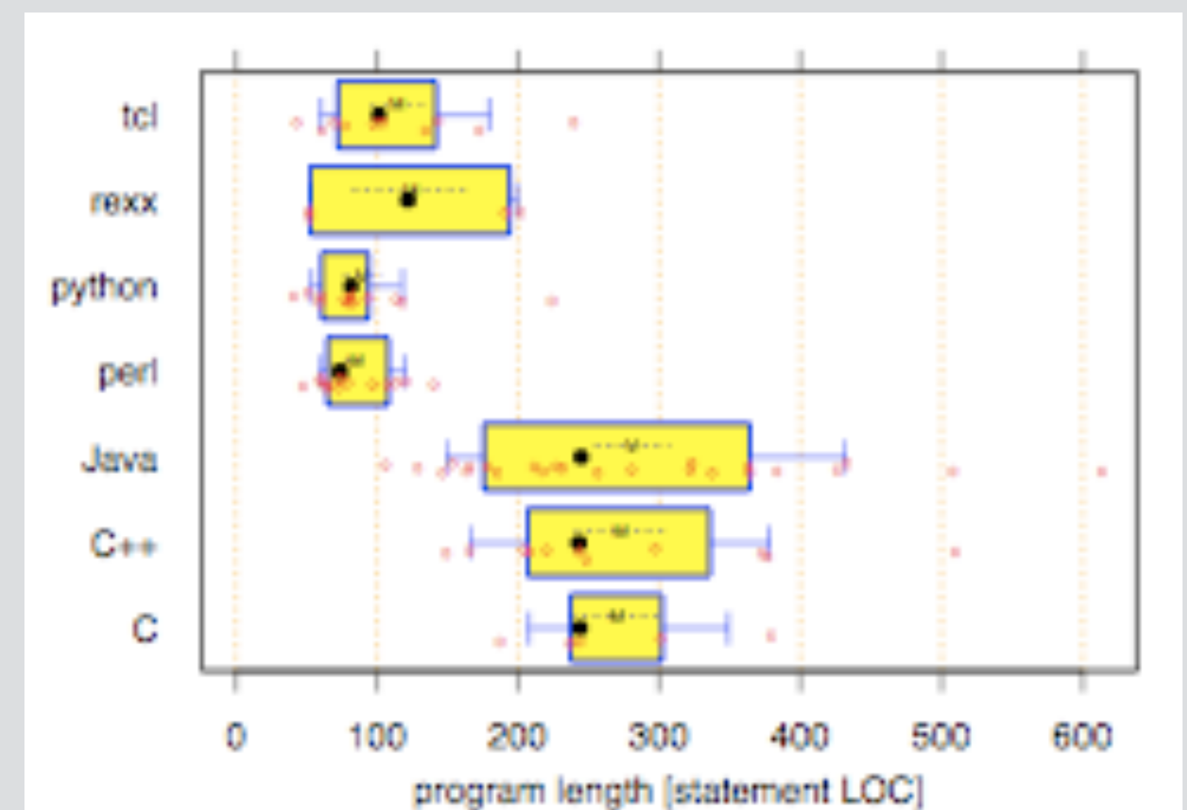


Figure 5: Program length, measured in number of non-comment source lines of code. The bad/good ratios range from 1.3 for C up to 2.1 for Java and 3.7 for Rexx. ••

Empirical results

- Speed & memory usage:

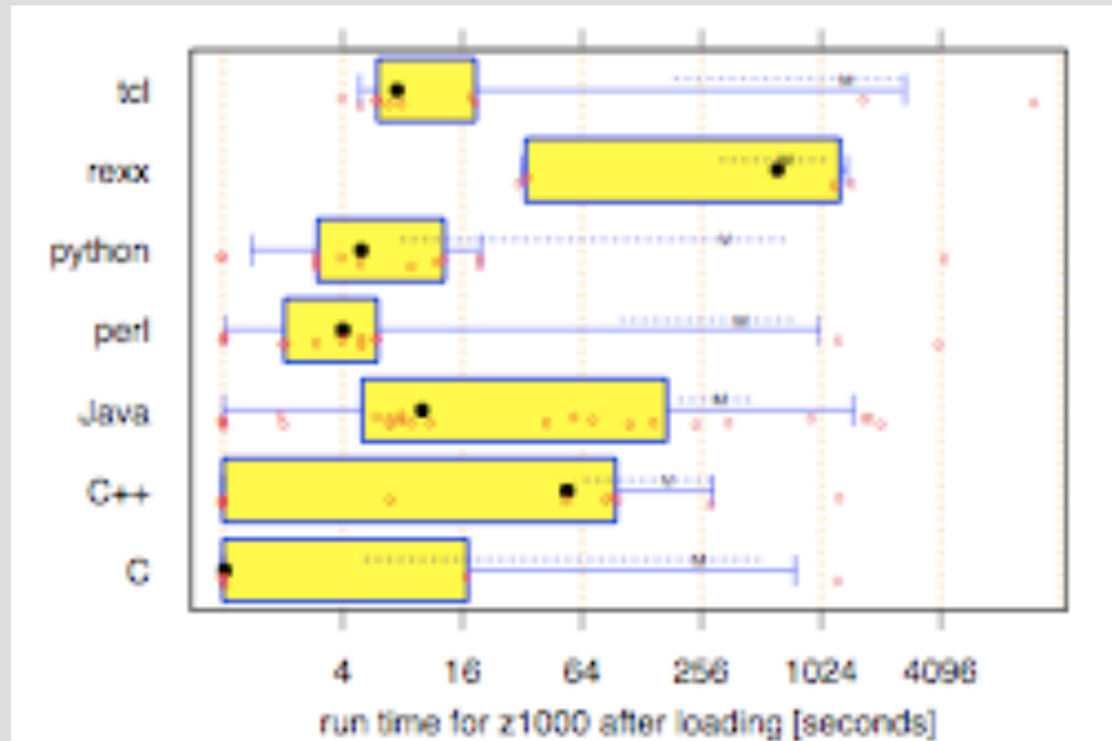


Figure 3: Program run time for the search phase only. Computed as time for z1000 data set minus time for z0 data set. Note the logarithmic axis. The bad/good ratios range from 2.9 for Perl up to over 50 for C++. ••

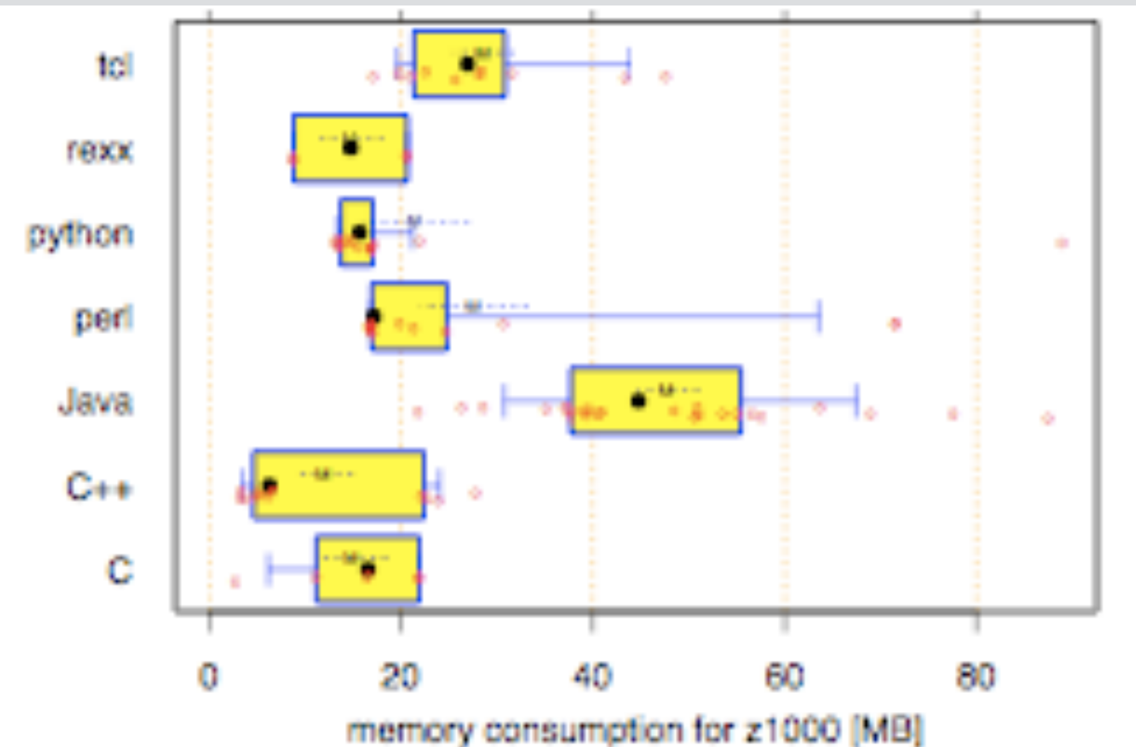


Figure 4: Amount of memory required by the program, including the interpreter or run time system, the program itself, and all static and dynamic data structures. The bad/good ratios range from 1.2 for Python up to 4.9 for C++. ••

Script execution

- Interpretes (typically in one pass) one after line
- during interpretations:
  1. validation (syntax)
  2. execution (semantics)

Conditionals result in skipping portions of the script
The skipped lines can contain wrong instructions for which syntactic analysis does happens until specific conditions are met.

In case of compiled languages

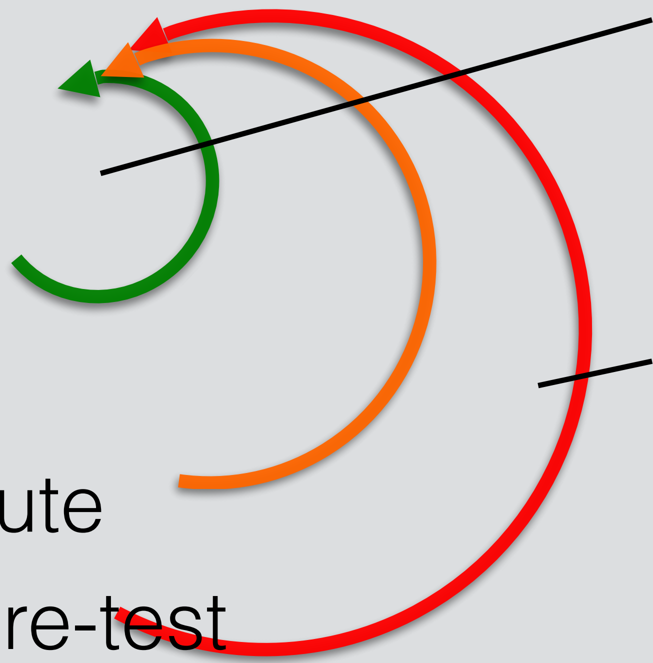
- Compiler – our friend & enemy at the same time
  - experienced programmer fixes compilation issue~2min.
    2 min x his/her salary is cost of fixing the issue (cheap)

- Not the only cycle to pass before the final product
  - next stages: testing -> debuting -> integration
  - longer
  - nonuniform

Bjarne Stroustrup - wikipedia

Cycles

- Compiled
  - edit
  - compile
  - test execute
  - debug & re-test



Cheapest cycle

The most expensive

- Scripts
  - edit
  - execute
  - debug

When not to use scripts

- Program = algorithms + data structures
- Not all scripting l. allow to have complex types (e.g.: bash text, arrays)
- Some algorithms difficult to implement (e.g.: bash has no floating point arithmetics)

Advisors

Be careful with people who know the best solution for your problem before they even bother to listen to the end.

Anybody who comes to you and says he has a perfect language is either naïve or a salesman.
in C++ 0x - An Overview at University of Waterloo Computer Science Club video

History

- JCL (Job Control language) IBM – mainframe →....→ shell (bash)
- SETL (lata 60’ NY) → ABC (80’ Netherlands) → Python(91 Netherlands)
- JavaScript (94’ Netscape)
  → ECMA (standard)
- ROOT/CINT (91’ CERN) → a trend caught by the industry as well

The big 6

- JavaScript – web apps. + natywny HTML5 + MS Windows + grafika
- Perl – text processing
- PHP – (LAMP) → Linux+Apache+MySql+PHP
- Python – idal glue
- TCL – pioneer
- VisualBasic w Excelu (the LOC champion) :-)

Other + DSL

- DOS batch - Windows system
- LUA
- AWK, sed
- DSL: HTML, SQL, YACC, Mathematica, matlabL

Programming patterns

- Patterns are like recipes:
  - Finding integrals: if there is polynomial in numerator and denominator ... do that and that
- Same for programming patterns:
  - If you need to build a complex object... use the builder pattern
  - Mostly discussed in OO domain (as things can easily go wrong there)
- Some of these patterns not applicable for scripts, some are not needed, there are additional ones

Categories of patterns

- Construction:
  - Singleton-, Factory-, Factory method+, Builder+, Prototype+
- Behavioural:
  - Chain of responsibility+, Message+, Interpreter ☺, Iterator-, Mediator+, Memento-, Observer+, Strategy+, Template method+, Visitor-,
- Structural:
  - Adapter ☺, Bridge-, Composite+, Decorator+, Flyweight+, Proxy+

Patterns in scripts

- Constructional patterns rarely used except if the construction of object/file/... is complex/repeatable —> good practice is to detach it from the rest of the script
- Bechavioural:
  - Decorator —> example pipelining:
    `cat dane | grep “xyz” | grep -v “ala” > filtered`
  - Chain of responsibility —> pipelining with split output
  - Message: scripts generation
  - Interpreter: oh yes, no effort if the same scripting language
  - Iterator: typically build in
  - Mediator, observer: rarely used
  - ...
  - Structural: rarely used (scripts operate on „simple” objects)

Patterns specific to scripting

- Data as a script! --- Active file
  - Typically the (partial) data saved in a format independent of the programming language used to produce/consume it
  - Standarised formats (e.g.: XML, binary) —> translation needed transient ←→ persistent
  - In case of scripts the data can be saved as **the script**:
    transient → persistent: may need generator
    persistent → transient: automatic

- Example, configuration file for the bash

```
source ~/.lab.conf
....
```

Patterns specific to scripting

- Bootstrap script
  - A variant of active file: input file reading with a step of conversion to the script
- Aliasing --- Command interceptor
  - When the behaviour needs to be extended (e.g.: rm -> mv ~/.trash)
  - When a simplification is needed: `alias ls="ls -ltr | sort -k5 -n"`
- Interpreter — invoking script in the script
  - very frequent and very useful

Summary

- Scripting languages are simpler from the system languages
  - Their predominant use is to serve as a glue to create larger systems
  - Irreplaceable in automation
  - Not suitable for all programming purposes (e.g.: numerics)

Next lecture

- Unix shell scripts in bash - text processing