

LEARNING MANAGEMENT SYSTEM TRAINING

VIRTUAL PROGRAMMING LAB

Hrvoje Leventić

hrvoje.leventic@ferit.hr

Osijek, June 13, 2019

AUTOMATIC GRADING OF PROGRAMMING ASSIGNMENTS

Often STDIN/STDOUT based workflow

Test cases

Online environment/compiler/IDE

Conform the code to the input and output

BENEFITS

Cross platform

Multilingual

Remotely accessible

Code tester



FERIT

WHY USE AN AUTOGRADER

Student engagement

Accelerates marking

More time with students

Easy to help students

WHAT AUTOGRADERS EXIST?

**There are a number of autograder options
out there**

Both commercial and free/open-source:

Notable mention:

repl.it



FERIT

WHY VPL 1/2

Allows students to run code in browser

Validates student code against teacher-designed test cases

Provides instant feedback to students

Autogrades student submissions

Already linked to student credentials

WHY VPL 2/2

Validates student code against other submissions (Plagiarism Checker)

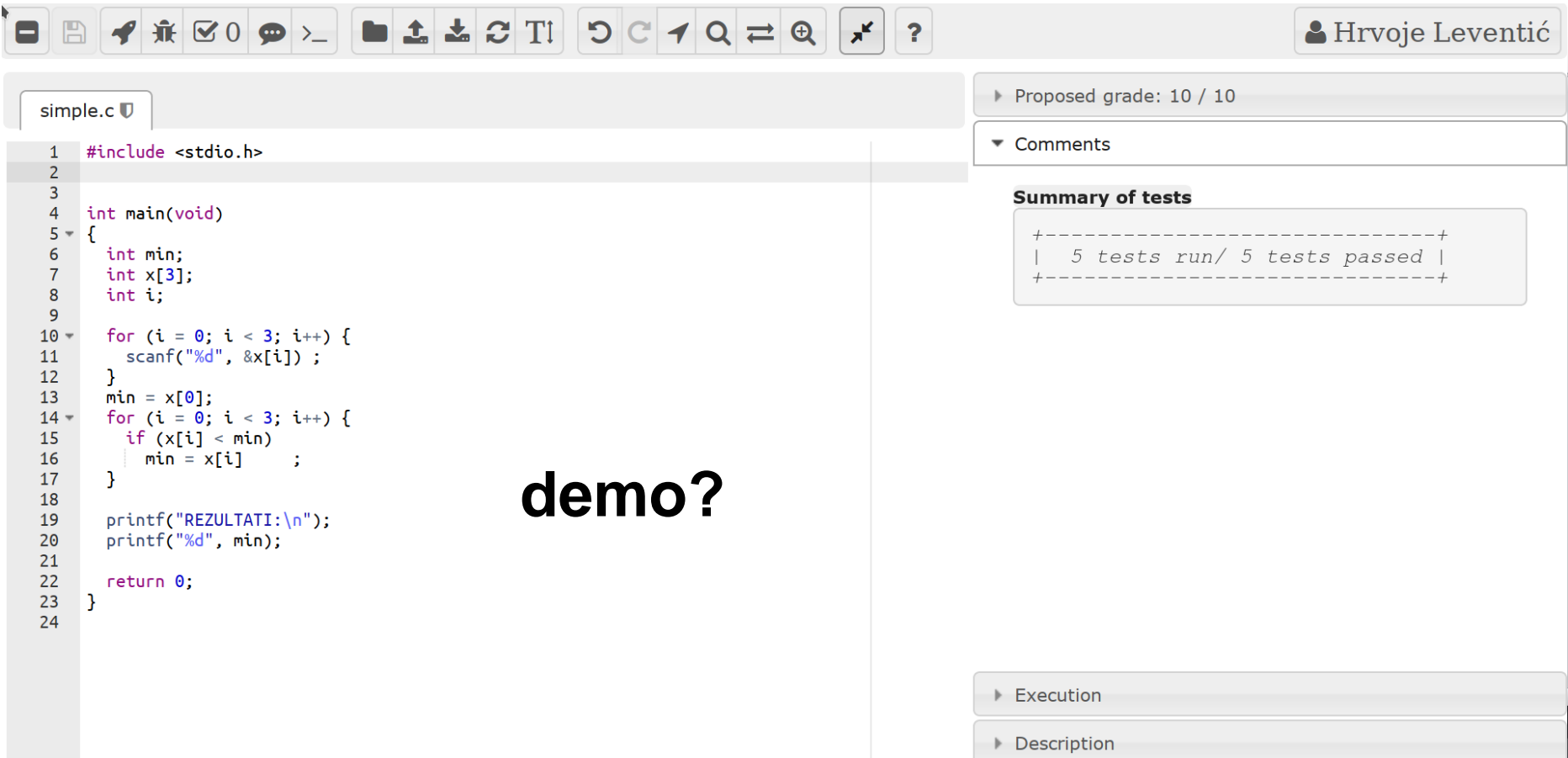
Grades go into Moodle Gradebook instantly

Transparency in Assessment

Support for multiple languages

Open source and in development since 2010

SO, HOW DOES IT WORK?



The image shows a screenshot of an IDE interface. At the top, there is a toolbar with various icons for file operations, editing, and navigation. The user's name, "Hrvoje Leventić", is displayed in the top right corner. The main area is divided into two panes. The left pane shows a C program named "simple.c" with the following code:

```
1 #include <stdio.h>
2
3
4 int main(void)
5 {
6     int min;
7     int x[3];
8     int i;
9
10    for (i = 0; i < 3; i++) {
11        scanf("%d", &x[i]);
12    }
13    min = x[0];
14    for (i = 0; i < 3; i++) {
15        if (x[i] < min)
16            min = x[i];
17    }
18
19    printf("REZULTATI:\n");
20    printf("%d", min);
21
22    return 0;
23 }
24
```

The right pane contains a "Proposed grade: 10 / 10" section, a "Comments" section, and a "Summary of tests" section. The "Summary of tests" section displays the following text:

```
+-----+
| 5 tests run/ 5 tests passed |
+-----+
```

At the bottom of the right pane, there are two sections: "Execution" and "Description".

demo?



ANATOMY OF A TEST CASE

Simple synthax

Four basic commands

Line matching

Exact matching or regex

vpl_evaluate.cases

```
1 case = first min
2 grade reduction = 20%
3 input = 1
4 12
5 99
6 output = /*RESULTS:
7 1/
8
9 case = second min minus
10 grade reduction = 20%
11 input = 10 -1 99
12 output = /*RESULTS:
13 -1/
14
15 case = third min minus
16 grade reduction = 20%
17 input = 68
18 99
19 -99
20 output = /*RESULTS:
21 -99/
22
```

DEMO 2

Simple line
matching

Two test cases

vpl_evaluate.cases

```
1 case = no newlines
2 grade reduction = 50%
3 input = 1 2 3
4 output = 1 2 3
5
6 case = newlines and whitespaces
7 grade reduction = 50%
8 input = 10
9 -1
10
11 99
12 output = 10
13 -1
14
15 99
16
```



DEMO 2

Add a simple python program

```
★ program.py ✕  
1 import sys  
2 import random  
3 print "filler"  
4 a = []  
5 for line in sys.stdin:  
6     a.append(line)  
7     #random.shuffle(a)  
8 for i in a:  
9     print i  
10    print "filler"  
11  
12  
13
```



FERIT

TEST CASES EASY TO FOOL

Test cases visible – easy to cheat:

Printing test cases

Hard to debug – tries to guess output

if number -> ignores text before

ignores whitespaces for numbers

numbers have to be in a correct order



FERIT

DEMO 3

See if you can fool the grader

DEMO 4

USE REGEX IN OUTPUT

- +Reduce cheating
- +Reduce accidental correct answers
- Students have to be careful about whitespace
- Constrain inputs
- Allow debug printing

```
vpl_evaluate.cases FT 32  
1 case = first  
2 grade reduction = 50%  
3 input = 6  
4 1  
5 2  
6 3  
7 4  
8 5  
9 6  
10 output = /.*RESULTS:  
11 1.*/  

```



MAIN VPL FUNCTIONS

The screenshot shows the VPL interface with several callout boxes pointing to specific features:

- Basic Test Case Building**: Points to the 'Description' tab and the task description.
- Evaluate/Debug/Autograde**: Points to the 'Submissions list' tab.
- Advanced Test Case Building**: Points to the 'Similarity' tab.
- Test Case Testing**: Points to the 'Test activity' tab.
- View Student Attempts**: Points to the 'Test activity' tab.
- Plagiarism Checker**: Points to the 'Test activity' tab.

The interface includes a navigation bar with tabs: Description, Submissions list, Similarity, and Test activity. The main content area displays a task description: "Task - A counter that skips every n th (loops)", available from Tuesday, 18 September 2018, 8:45 AM. It specifies a maximum of 1 file, individual work, no grade, and that it can be run and evaluated. The task requires a program that outputs numbers skipping every n th, with an example sequence: 1-2-4-5-7-8-10-11... and instructions to use `$input = trim(fgets(STDIN));` for input gathering.

Execution files are listed as `vpl_evaluate.cases`. A code block shows the following test cases:

```
1- case = Test 1
2 input = 3
3 output = 1245781011131416171920
4
5- case = Test 2
6 input = 7
7 output = 1234568910111213151617181920
8
9- case = Test 3
10 input = 5
11 output = 123467891112131416171819
12
13- case = Test 4
14 input = 2
15 output = 135791113151719
16
17- case = Test 5
```

A settings menu is visible on the right side, containing options such as Edit settings, Test cases, Execution options, Requested files, Advanced settings, Execution files, Maximum execution resources limits, Files to keep when running, Variations, Check execution servers, Local execution servers, Test activity, Submission, Edit, Submission view, Grade, Previous submissions list, Virtual programming labs, Locally assigned roles, Permissions, Check permissions, and Filters.

DEMO 5

Solve the Demo 5 exercise as we will use it to showcase:

similarity checking

automatic grading

commenting on grades

previous versions

direct access to students code



FERIT

REQUESTED FILES

Great for providing boilerplate code to students

Force a certain language:

1 Create requested files with the desired extensions

2 Limit number of files to the number of your files

3 Students won't be able to delete them upon evaluation



EXECUTION FILES

Used to compile and evaluate student code

Not visible to students

vpl_evaluate.cases one of them

Available at compile time:

- **Great for libraries students should not be able to change**
- **Overwrites student's file if same name**

DESIGN YOUR OWN VPL ACTIVITY

Create new VPL activity

Configure Execution options

Write Test cases

**Add Requested files if you want the provide
a boilerplate code for your students**

Test the activity



FERIT

TIPS AND TRICKS

Scanf vs. gets vs. fgets

Turn off stdout buffering

```
setvbuf(stdout, NULL, _IONBF, 0);
```

Safe exam browser

Duplicating activity does not duplicate grades – past exams free for practice

Creative cheaters – cheatsheet, grep

EVEN THIS MUCH CAN TAKE YOU QUITE FAR

**SOME CREATIVITY, LARGE TEST CASES
AND MANUALLY CHECKING STUDENTS'
CODE SOLVES MOST OF THE PROBLEMS**

**OFTEN – IT'S THE EASIER AND LESS TIME
CONSUMING OPTION**



Lets Go Further

PREVENTING CHEATING WITH TWO SETS OF TEST CASES

Often used to prevent cheating by simple printing of the test cases

One set of test cases visible to students

Other set uploaded after deadline and regraded

Program has to work for both test cases

+Easy +Fast +NoSkillRequired™

- Hard to make test cases comparable and prevent edge cases

DYNAMIC TEST CASES

„These are too many students, I cannot possibly check every submission manually, let me automate this”

The Holy Grail of student cheating prevention

+Prevents cheating by printing the testcases as they are regenerated on each evaluation and different on every run

+It makes you look cool

+You don't have to check the code of each submission manually

-It's hard to do properly -It's time consuming

-You have to know a bit of linux and bash

VPL CODE EXECUTION LIFECYCLE

Three stages:

1. **Compilation** – Moves student code to server; `vp1_run.sh/vp1_debug.sh` creates `vp1_execution`
2. **Running** – Runs `vp1_execution` in execution jail with input from `vp1_evaluate.cases`
3. **Evaluation** – `vp1_evaluate.sh` parses output, calculates score

VPL EXECUTION FILES

`vp1_run.sh/vp1_debug.sh` – prepares for run, generates executable

`vp1_execution` – the executable, runs in jail, does not have access to other files

`vp1_evaluate.sh` - runs the executable, provides input, collects output, generates the grades according to testcases

`vp1_evaluate.cases` – the testcases

HIJACKING ONE OF THE EXECUTION STAGES

Two approaches:

1. Hijack `vp1_evaluate.sh` – write your own; provide input to executable, parse output, generate result text (**weird synthax, hard, edge cases**)
2. Hijack `vp1_run.sh` – insert your own code to run before creating the executable file, directly generate `vp1_evaluate.cases` file (**easier to generate testcases, access to student code before compilation**)

WRITING YOUR OWN EVALUATION LOGIC

Makes sense when the program output is not STDOUT

We used it to check writing to binary files

-The most buggy, most complained about lab exercise

+Lots of possibilities, e.g. Sending the file with curl somewhere else

Testing 1/2 : first

Testing 2/2 : newlines and whitespaces

<|--

-Failed tests

Test 1: first

Test 2: newlines and whitespaces

--|>

<|-- -Test 1: first (-50.000)

Incorrect program output

--- Input ---

> 6

>1

>2

>3

>4

>5

>6

--- Program output ---

> 6

>

>1

>

>2

>

>3

>

>4

>

>5

>

--- Expected output (regular
expression)---

>.*RESULTS:

>1.*

DYNAMICALLY GENERATING TESTCASES



Easier to accomplish

**Benefit from a very well implemented
evaluator logic**

**Testcase file synthax easy to generate
dinamically**

Pre-compilation checks

Free to use any language



FERIT

OUR BEST APPROACH

vpl_run.sh:

```
#load common script and check programs
```

```
. common_script.sh
```

```
check_program gcc
```

```
get_source_files c
```

```
python generator.py
```

```
#compile
```

```
eval gcc -o vpl_execution -std=c99 $SOURCE_FILES  
-lm -lutil
```



FERIT

OUR EXPERIENCE SHOWCASE AND IDEAS

Programming 2 exercises

Students only use C

Generators mostly in python

It's hard to create fun exercises in C



FERIT

DEMO

Build your own dynamic testcase generator