C-Pack of IPAs: A C90 Program Benchmark of Introductory Programming Assignments

Pedro Orvalho
pmorvalho@tecnico.ulisboa.pt
INESC-ID, IST, U. Lisboa
Lisbon, Portugal

Mikoláš Janota
mikolas.janota@cvut.cz
Czech Technical University in Prague
Prague, Czech Republic

Vasco Manquinho
vasco.manquinho@tecnico.ulisboa.pt
INESC-ID, IST, U. Lisboa
Lisbon, Portugal

ABSTRACT
Given the vast number of students enrolled in Massive Open Online Courses (MOOCs), there has been a notable surge in automated program repair techniques tailored for introductory programming assignments (IPAs). These techniques leverage correct student implementations to provide automated, comprehensive, and personalized feedback to the students.

This paper presents C-Pack-IPAs, a publicly available benchmark comprising student-program submissions for 25 distinct IPAs. C-Pack-IPAs contains semantically correct, semantically incorrect, and syntactically incorrect programs, along with a dedicated test suite for each IPA. Hence, C-Pack-IPAs serves as a valuable resource for evaluating the progress of novel automated program repair frameworks, addressing both semantic and syntactic aspects, with a specific focus on providing feedback to novice programmers.

Notably, some semantically incorrect programs in C-Pack-IPAs have been manually fixed and annotated with diverse program features, enhancing their utility for the development of various program analysis frameworks. Moreover, we conducted evaluations on C-Pack-IPAs using two leading semantic program repair tools tailored for IPAs, CLARA and VERIFIX.

CCS CONCEPTS
• Applied computing → Computer-assisted instruction; • Theory of computation → Program semantics; Program analysis; Program reasoning; • Computing methodologies → Machine learning.

KEYWORDS
Introductory Programming Assignments, Programming Education, Automated Program Repair, Semantic Program Repair, Syntactic Program Repair, Computer-Aided Education

1 INTRODUCTION
Nowadays, thousands of students enroll every year in programming-oriented university courses and in Massive Open Online Courses (MOOCs) [4]. Providing feedback to novice students in introductory programming assignments (IPAs) in these courses requires substantial effort and time by the faculty. Hence, there is an increasing need for systems that offer automated, comprehensive, and personalized feedback to students with incorrect submissions in programming assignments.

Typically, in Computer Science courses, a programming assignment follows a familiar pattern: the lecturer outlines a computational problem, students devise solutions, and each solution undergoes evaluation for correctness using predetermined tests. If the students’ tentative solutions do not pass a given test, they are deemed incorrect without helpful feedback. In instances where students’ programs fail a subset of the predefined tests, seeking feedback from the lecturer becomes a common practice to understand the reasons behind the unexpected behavior. When a program fails to pass even a single predefined test, it signifies a semantic error in the implementation. Unfortunately, due to the escalating number of student enrollments, personalized feedback from the faculty may not always be feasible. Therefore, automated semantic program repair frameworks [1, 4, 6, 7, 10–12, 18, 20, 22] are ideal for providing hints on how students should repair their incorrect programming assignments.

This paper presents C-Pack-IPAs, a C90 Program benchmark of introductory programming assignments (IPAs). C-Pack-IPAs comprises students’ programs submitted for 25 different IPAs along with the respective test suite used for each assignment. The details of the IPAs are provided in Section 3. For each IPA, C-Pack-IPAs includes sets of both semantically correct and incorrect implementations. Additionally, C-Pack-IPAs encompasses a collection of syntactically faulty programs submitted for each IPA. The primary objective of this paper is to present C-Pack-IPAs, a resource featuring semantically and syntactically incorrect student implementations. This benchmark is intended to facilitate the evaluation of novel automated program repair frameworks, addressing both semantic and syntactic aspects, with a focus on assisting novice programmers.

C-Pack-IPAs includes a subset of semantically incorrect programs, manually annotated with various features such as the number of faults in each program, the location of these faults, and the type of each fault. Additionally, C-Pack-IPAs also comprises corrected versions of these programs, which were manually fixed. The availability of annotated and fixed programs serves as a valuable resource, allowing developers to validate their results during the development of novel program analysis tools, particularly for tasks such as program repair and fault localization.
Table 1: High-level Description of C-Pack-IPAs Benchmark.

<table>
<thead>
<tr>
<th>Labs</th>
<th>#IPAs</th>
<th>#Correct Submissions</th>
<th>#Semantically Incorrect Submissions</th>
<th>#Syntactically Incorrect Submissions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lab02</td>
<td>10</td>
<td>799</td>
<td>486</td>
<td>223</td>
</tr>
<tr>
<td>Lab03</td>
<td>7</td>
<td>351</td>
<td>699</td>
<td>106</td>
</tr>
<tr>
<td>Lab04</td>
<td>8</td>
<td>465</td>
<td>246</td>
<td>119</td>
</tr>
<tr>
<td>Total</td>
<td>25</td>
<td>1615</td>
<td>1434</td>
<td>448</td>
</tr>
</tbody>
</table>

The main contributions of this work are:
- C-Pack-IPAs, a benchmarks consisting of C programs submitted for 25 different IPAs, during three academic years. C-Pack-IPAs contains semantically correct, semantically incorrect, and syntactically incorrect programs plus a test suite for each IPA.
- A subset of C-Pack-IPAs’s semantically incorrect programs has been manually fixed and annotated, incorporating several program features such as the number of faults and the locations of these faults.
- The evaluation of C-Pack-IPAs was conducted using two state-of-the-art program repair tools tailored for IPAs.
- C-Pack-IPAs is going to be publicly available on GitHub: https://github.com/pmorvalho/C-Pack-IPAs.

The structure of the remainder of this paper is as follows. Section 2 presents C-Pack-IPAs. Next, Section 3 presents a brief description of the set of programming exercises. Section 4 presents the experimental evaluation where we evaluated C-Pack-IPAs using state-of-the-art program repair tools. Lastly, Section 5 describes related work, and the paper concludes in Section 6.

2 C-Pack-IPAs

C-Pack-IPAs is a pack of student programs developed during an introductory programming course in the C programming language. These programs were collected over three distinct practical classes for 25 different IPAs at Instituto Superior Técnico, throughout three academic years. The set of submissions was split into three groups: semantically correct, semantically incorrect, and syntactically incorrect submissions. The students’ submissions that satisfied the set of input-output test cases for each IPA were considered semantically correct. The submissions that failed at least one input-output test but successfully compiled were considered semantically incorrect implementations. Lastly, the students’ submissions that did not successfully compile were considered syntactically incorrect.

Table 1 presents the number of submissions gathered. For 25 different programming exercises, this benchmark contains 1615 different correct programs, 1434 semantically incorrect submissions, and 448 syntactically incorrect implementations. C-Pack-IPAs is organized chronologically. This arrangement proves valuable for training and evaluating new program analysis tools. For instance, the programs from the first academic year can serve as training data, those from the second year as the validation set, and the programs from the third year as the evaluation set. C-Pack-IPAs is organized.

Appendix A presents three tables with the number of student submissions (correct, semantically incorrect and syntactically incorrect) received for each of the 25 different programming assignments.

Furthermore, C-Pack-IPAs only contains students’ submissions that gave their permission to use their programs for academic purposes. Each student’s identification was anonymized for privacy reasons, and all the comments were removed from their programs. A unique identifier was assigned to each student. These identifiers are consistent among different IPAs and different years of the programming course. For example, if the identifier stu_15 appears in more than one programming exercise, it corresponds to the same student. If some students take the course more than once, they are always assigned to the same anonymized identifier. Currently, C-Pack-IPAs contains submissions from 102 different students.

Annotated Programs. A benchmark of programs should be enriched with informative annotations, enabling developers to validate their results while developing novel program analysis tools for tasks such as program repair and fault localization. With this objective in mind, we have meticulously annotated the Lab 02 submissions in C-Pack-IPAs with various program features. These annotations serve as valuable resources for developers and facilitate the training and evaluation of machine learning models. Each semantically incorrect program in C-Pack-IPAs’s Lab 02 submissions has been annotated with the following features:
- #Variable: indicates the number of different variables present in each program;
- Program Features: encompasses various program features, including uninitialized variables, segmentation faults, etc., providing valuable information for analysis;
- #Passed Tests: represents the count of passed tests from the test suite;
- #Failed Tests: specifies the number of failed tests from the test suite;
- IO tests’ output: presents the output obtained by running the incorrect program with the test suite;
- #Faults: indicates the total number of faults present in the program;
- Faults: enumerates the list of faulty instructions/expressions within the program;
- Faulty Lines: lists the program lines containing faults;
- Faults’ Types: specifies the types of each fault in the program;
- Repair Actions: enumerates the required repair actions to fix the faulty instructions/expressions, with possible actions including Insert, Replace, Remove, or Move;
- Suggested Repairs: provides a list of suggested repairs for each identified faults in the program;
- Next Correct Submission: indicates the path to the subsequent correct submission by the same student, if available.

Table 2 presents the various types of faults used to annotate programs in C-Pack-IPAs. For each manually annotated fault type, the table presents the count of faulty programs exhibiting that specific fault, categorized by each exercise of Lab 02. The most prevalent fault types include Presentation Error (differences only in white spaces), Incorrect Output (output is incorrect), and Uninitialized Variable. Additionally, at the bottom of Table 2, the average number of faults in programs for each exercise of Lab 02 is provided. All annotations are stored in two formats: as plain text (txt) files and within an SQLite3 database.
Table 2: The number of faulty programs in each exercise of Lab02, with different types of program faults.

<table>
<thead>
<tr>
<th>Fault Type</th>
<th>Lab 02</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Ex 01</td>
</tr>
<tr>
<td>Incomplete Binary Operation</td>
<td>1</td>
</tr>
<tr>
<td>Incorrect Data Type</td>
<td>4</td>
</tr>
<tr>
<td>Incorrect Input</td>
<td>63</td>
</tr>
<tr>
<td>Incorrect Output</td>
<td>1</td>
</tr>
<tr>
<td>Misplaced Expression</td>
<td>1</td>
</tr>
<tr>
<td>Misplaced Loop Decrement</td>
<td>1</td>
</tr>
<tr>
<td>Missing Expression</td>
<td>2</td>
</tr>
<tr>
<td>Missing Instruction</td>
<td>8</td>
</tr>
<tr>
<td>Missing Instructions</td>
<td>5</td>
</tr>
<tr>
<td>Missing Loop Decrement</td>
<td>1</td>
</tr>
<tr>
<td>Missing Loop Increment</td>
<td>1</td>
</tr>
<tr>
<td>Missing Output</td>
<td>1</td>
</tr>
<tr>
<td>Non-zero Return</td>
<td>1</td>
</tr>
<tr>
<td>Presentation Error</td>
<td>44</td>
</tr>
<tr>
<td>Uninitialized Variable</td>
<td>11</td>
</tr>
<tr>
<td>Variable Misuse</td>
<td>2</td>
</tr>
<tr>
<td>Wrong Binary Operation</td>
<td>3</td>
</tr>
<tr>
<td>Wrong Comparison Operator</td>
<td>4</td>
</tr>
<tr>
<td>Wrong Exercise</td>
<td>8</td>
</tr>
<tr>
<td>Wrong Expression</td>
<td>7</td>
</tr>
<tr>
<td>Wrong Initialization</td>
<td>1</td>
</tr>
<tr>
<td>Wrong Instruction</td>
<td>5</td>
</tr>
<tr>
<td>Wrong Literal</td>
<td>1</td>
</tr>
<tr>
<td>Wrong Parameter</td>
<td>1</td>
</tr>
<tr>
<td>Average Number of Faults</td>
<td>1.8</td>
</tr>
</tbody>
</table>

Organization. C-Pack-IPAs is structured by lab and exercise, organized chronologically by academic year. Each program is stored in its respective folder, along with the program’s output results for the test suite and the previously described handmade annotations.

3 IPAS DESCRIPTION

The set of IPAs corresponds to three different lab classes of the introductory programming course to the C programming language. Each lab class focuses on a different topic of the C programming language. In Lab02, the students learn how to program with integers, floats, IO operations (mainly printf and scanf), conditionals (if-statements), and simple loops (for and while-loops). In Lab03, the students learn how to program with loops, nested loops, auxiliary functions, and chars. Finally, in Lab04, the students learn how to program with integer arrays and strings. The textual description of each programming assignment can be found in the public GitHub repository, and the input/output tests used to evaluate semantically the set of students’ submissions. Moreover, there is also a reference implementation for each IPA in the public git repository that can be used by program repair frameworks that only accept a single reference implementation to repair incorrect programs. Appendix A presents the entire list of IPAs.

4 EXPERIMENTAL RESULTS

To evaluate C-Pack-IPAs, we used two publicly available state-of-the-art program repair tools for fixing introductory programming assignments (IPAs): CLARA [4] and VERIFIX [1]. We simply focused on the set of semantically incorrect programs which is composed by 1434 programs as presented in Table 1.

CLARA and VERIFIX. VERIFIX [1] aligns the control flow graph (CFG) of an incorrect program with the reference solution’s CFG. Then, using that alignment relation and MaxSMT solving, VERIFIX proposes fixes to the incorrect program. VERIFIX also requires a compatible control flow graph between the incorrect and the correct program. On the other hand, to repair an incorrect program, CLARA [4] receives either one or a set of correct programs. This set of programs corresponds to clusters’ representatives produced by CLARA. During CLARA’s repair process, if none of the correct programs provided has an exact match with the incorrect submission’s control flow, then CLARA is not able to repair the program and returns a Structural Mismatch error. Otherwise, CLARA gathers the set of repairs using each correct program and returns the minimal one. Since CLARA can take advantage of several correct implementations from previous years for a given IPA, we fed CLARA all correct programs, from the three different academic years, to generate clusters for each IPA in our benchmark. Furthermore, we also run CLARA using the faculty’s reference implementation for each IPA, i.e., CLARA uses a single program and not a set of clusters’ representatives. Moreover, we run VERIFIX using also the reference implementation since VERIFIX can only accept a single correct program as input.

Experimental Setup. All the experiments were conducted on an Intel(R) Xeon(R) Silver computer with 4210R CPUs @ 2.40GHz, using a memory limit of 32GB and a timeout of 600 seconds.

Results. Table 3 presents the number of programs repaired by VERIFIX, CLARA (utilizing a single reference implementation), and CLARA (utilizing its own clusters). The results indicate that VERIFIX demonstrates success primarily in Lab 02, repairing approximately 19% of the programs. However, VERIFIX encounters challenges in repairing 80% of Lab 02 and the entire set of programs from Lab 03 and Lab 04. The primary factor influencing VERIFIX’s performance is its limited support for certain C Library functions utilized in several exercises. For instance, nearly all exercises in Lab 03 involve the use of C Library functions such as putchar or getchar, which are not supported by VERIFIX.
Table 3: The number of programs repaired by Verifix, CLARA without using clusters and CLARA using clusters.

<table>
<thead>
<tr>
<th>Repair Method</th>
<th>% Fixed</th>
<th>% Structural Mismatch</th>
<th>% Unsupported Features</th>
<th>% Other Errors/Exceptions</th>
<th>% Timeouts (600s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLARA (No Clusters)</td>
<td>11 (1.57%)</td>
<td>0 (0.0%)</td>
<td>699 (100.0%)</td>
<td>0 (0.0%)</td>
<td>0 (0.0%)</td>
</tr>
<tr>
<td>CLARA (Clusters)</td>
<td>188 (19.74%)</td>
<td>328 (46.92%)</td>
<td>138 (19.74%)</td>
<td>3 (1.22%)</td>
<td>0 (0.0%)</td>
</tr>
</tbody>
</table>

As previously mentioned, if none of the correct programs provided matches the control flow of the incorrect submission exactly, CLARA issues a \textit{structural mismatch} error. Table 3 reveals that CLARA, when employing a reference implementation, exhibits a notably higher percentage of structural mismatch errors compared to CLARA utilizing clusters. This difference arises because clusters, with multiple programs, offer various control flow options, leading to a reduced rate of structural mismatches. Additionally, CLARA generates a set of repairs for each cluster’s representative. Therefore, a higher number of clusters corresponds to more time spent in the repair process. This constitutes one of the primary reasons for the increased occurrence of timeouts observed with CLARA when utilizing clusters as opposed to not using clusters.

Figure 1 presents a cactus plot illustrating the CPU time allocated for repairing each program (on the x-axis) in relation with the number of successfully repaired programs (on the y-axis) across the three different repair techniques. The legend is organized in descending order based on the count of programs successfully repaired. Notably, VERIFIX, within the 60-seconds, repairs 90 out of 1434 programs (approximately 6.5%). In comparison, CLARA, utilizing a reference implementation, repairs 286 programs (20%), while employing its own clusters allows CLARA to repair around 500 programs within the same time limit (approximately 35%).

Figure 2 illustrates a scatter plot comparing the CPU time spent using CLARA’s clusters against running CLARA with the faculty’s reference implementation (no clusters). Each data point in the plot represents a program, where the x-value (corresponding to using CLARA’s clusters) and y-value (representing no clusters) denote the CPU time spent on repairing that program.

If a point falls below the diagonal, it indicates that using a reference implementation outperformed using clustering. In this scenario, CLARA, with its own clusters, repairs each program more slowly than with a single correct program. Consequently, when considering programs repaired by both clustering methods, not using clusters is faster, although repairs a significantly smaller number of programs, as presented in Table 3. Additionally, Figure 2 highlights that the set of programs repaired by CLARA differs when using a reference implementation compared to clusters.

In short, we evaluated two state-of-the-art semantic program repair tools tailored for IPAs. CLARA was the clear winner, repairing 550 programs equivalent to 38.4% of C-Pack-IPAs. This outcome indicates ample room for improvement. Given that C-Pack-IPAs encompasses 25 distinct IPAs of varying complexities, it stands as a valuable resource for the development of advanced program repair tools capable of addressing more intricate IPAs. Moreover, C-Pack-IPAs [14] has also proven successful in evaluating various works across program analysis [16, 17], program transformation [13], and program clustering [15].

5 RELATED WORK

Over the last few years, several program repair tools [4–6, 20] have exploited diverse correct implementations from previously enrolled students for each IPA to repair new incorrect student submissions. On the one hand, some syntactic program repair tools [5, 21] have been developed to help students with compilation errors. On the other hand, semantic program repair has also been used to help repair students’ programs semantically [1, 4, 6, 9, 20]. However, the number of publicly available benchmarks to help develop and evaluate new program repair tools is significantly small [13]. The ITSP dataset [22] has been used by other automated software repair tools [1, 22] that use only one reference implementation. This dataset is also a collection of C programs although it is well balanced, i.e., the number of correct submissions is closer to the number of incorrect submissions in this dataset. The INTROCLASS dataset [8] is a collection of C programs submitted to six different IPAs and has the information about the number of defects in each program and the total number of unique defects for each IPA. CODEFLAWS [19] is a dataset of programs submitted for programming competitions.
C-Pack-IPAs: A C90 Program Benchmark of IPAs

6 CONCLUSION

C-Pack-IPAs, a C90 Program benchmark of introductory programming assignments (IPAs), is publicly available benchmark of students' submissions for 25 different programming assignments. C-Pack-IPAs has a set of semantically correct and incorrect implementations as well as syntactically faulty programs submitted for each IPA. To the best of our knowledge, C-Pack-IPAs is one of the few, if not the only, benchmark of IPAs written in the C programming language that contains both semantically and syntactically incorrect students' implementations and diverse correct implementations for the same IPA. Thus, C-Pack-IPAs can help evaluate novel semantic, as well as syntactic, automated program repair frameworks whose goal is to assist novice programmers in introductory programming courses. We have also manually fixed and annotated some of C-Pack-IPAs's semantically incorrect programs with several program features to help developing all sort of program analysis frameworks. Additionally, we evaluated C-Pack-IPAs using two state-of-the-art semantic program repair tools tailored for IPAs, Clara and Verifix.

ACKNOWLEDGEMENTS

This work was partially supported by Portuguese national funds through FCT, under projects UIDB/50021/2020 (DOI: 10.54499/UIDB/50021/2020), PTDC/CCI-COM/2156/2021 (DOI: 10.54499/PTDC/CCI-COM/2156/2021) and 2022.03.353. PTDC (DOI: 10.54499/2022.03.353. PTDC) and grant SFH/B/07724/2020. This work was also supported by the European Regional Development Fund under the Czech project AI&Reasoning no. CZ.02.1.01/0.0/0.0/15_003/0000466 (JP), and by the MEYS within the program ERC CZ under the project POSTMAN no. LL1902 co-funded by the European Union under the project ROBOPROX (reg. no. CZ.02.01.01/00/22_008/0004590). This article is part of the RICAIP project that has received funding from the EU’s Horizon 2020 research and innovation program under grant agreement No 857306.

REFERENCES


1 https://program-repair.org/benchmarks.html
A NUMBER OF SUBMISSIONS

Table 4 presents the number of semantically correct student submissions received for 25 different programming assignments over three lab classes for three different years. Next, Table 5 presents the set of syntactically incorrect submissions, while Table 6 describes the set of syntactically incorrect programs.

B LIST OF INTRODUCTORY PROGRAMMING ASSIGNMENTS (IPAS)

B.1 Lab02 - Integers and IO operations.

In Lab02, the students learn how to program with integers, floats, IO operations (mainly printf and scanf), conditionals (if-statements), and simple loops (for and while-loops).

IPA #1: Lab02 - Ex01. Write a program that determines and prints the largest of three integers given by the user.

IPA #2: Lab02 - Ex02. Write a program that reads two integers ‘N’, ‘M’, and prints the smallest of them in the first row and the largest in the second.

IPA #3: Lab02 - Ex03. Write a program that reads two positive integers ‘N’, ‘M’ and prints ‘yes’ if ‘M’ is a divisor of ‘N’, otherwise prints “no”.

IPA #4: Lab02 - Ex04. Write a program that reads three integers and prints them in order on the same line. The smallest number must appear first.

IPA #5: Lab02 - Ex05. Write a program that reads a positive integer ‘N’ and prints the numbers ‘1…N’, one per line.

IPA #6: Lab02 - Ex06. Write a program that determines the largest and smallest number of ‘N’ real numbers given by the user. Consider that ‘N’ is a value requested from the user. The result must be printed with the command ‘printf(‘min: %f, max: %f
’, min, max).’ Hint: initialize the largest and smallest to the first read value.

IPA #7: Lab02 - Ex07. Write a program that asks the user for a positive integer ‘N’ and prints the number of divisors of ‘N’. Remember that prime numbers have 2 divisors.

IPA #8: Lab02 - Ex08. Write a program that calculates and prints the average of ‘N’ real numbers given by the user. The program should first ask the user for an integer ‘N’, representing the number of numbers to be entered. The real numbers must be represented by float type. The result must be printed with the command ‘printf(‘%.2f
’, avg)’.

IPA #9: Lab02 - Ex09. Write a program that asks the user for a value ‘N’ corresponding to a certain period of time in seconds. The program should output this period of time in the format ’HH:MM:SS’. Hint: use the operator that calculates the remainder of division (‘%’).

IPA #10: Lab02 - Ex10. Write a program that asks the user for a positive value ‘N’. The output should present the number of digits that make up ‘N’ (on the first line), as well as the sum of the digits of ‘N’ (on the second line). For example, the number 12345 has 5 digits and the sum of these digits is 15.

B.2 Lab03 - Loops and Chars.

In this lab, the students learn how to program with loops, nested loops, auxiliary functions and chars.

IPA #11: Lab03 - Ex01. Write a program that draws a square of numbers like the following using the function ‘void square(int N);’.
Table 4: The number of semantically correct student submissions received for 25 different programming assignments over three lab classes for three different years.

<table>
<thead>
<tr>
<th>Year</th>
<th>Lab02</th>
<th>Lab03</th>
<th>Lab04</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>22 22 18 18</td>
<td>22 18 18 18</td>
<td>22 18 18 18</td>
<td>84</td>
</tr>
<tr>
<td>3</td>
<td>13 8 6 2</td>
<td>6 5 3 1</td>
<td>8 7 6 3</td>
<td>27</td>
</tr>
</tbody>
</table>

Table 5: The number of semantically incorrect student submissions received for 25 different programming assignments over three lab classes for three different years.

<table>
<thead>
<tr>
<th>Year</th>
<th>Lab02</th>
<th>Lab03</th>
<th>Lab04</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>36 10 7 2</td>
<td>32 30 20 6</td>
<td>3 11 5 6</td>
<td>108</td>
</tr>
<tr>
<td>2</td>
<td>28 2 1 1</td>
<td>14 10 17 15</td>
<td>9 1 1 1</td>
<td>46</td>
</tr>
<tr>
<td>3</td>
<td>51 43 33 33</td>
<td>58 78 14 14</td>
<td>14 10 17 15</td>
<td>215</td>
</tr>
</tbody>
</table>

Table 6: The number of syntactically incorrect student submissions received for 25 different programming assignments over three lab classes for three different years.

<table>
<thead>
<tr>
<th>Year</th>
<th>Lab02</th>
<th>Lab03</th>
<th>Lab04</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12 1 1 1</td>
<td>6 3 1 6</td>
<td>2 1 1 2</td>
<td>21</td>
</tr>
<tr>
<td>2</td>
<td>6 3 0 5</td>
<td>1 0 0 1</td>
<td>1 1 1 1</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>24 17 14 20</td>
<td>15 10 11 11</td>
<td>5 4 3 2</td>
<td>114</td>
</tr>
</tbody>
</table>

2. The tab (character "\t") must be used as the separator. The square shown is the example for 'N = 5'.

IPA #12: Lab03 - Ex02. Write a program that draws a pyramid of numbers using the ‘void pyramid(int N);’ function. The value of 'N', given by the user, must be greater than or equal to 2. The space (character "\t") must be used as the separator. The pyramid shown is the example for 'N = 5'.

IPA #13: Lab03 - Ex03. Write a program that draws a cross on diagonals using the ‘void cross(int N);’ function. The asterisk ('*' character) must be used to draw the cross; hyphen ('-' character) must be used as the separator. The crosses shown are the examples for 'N = 3' and 'N=8'.

7
**IPA #14: Lab03 - Ex04.** Write a program that reads a sequence of numbers separated by spaces and newlines, and print the same string, but the numbers in the output should not contain 0 at the beginning, eg `007` should print `7`. The exception is the number 0, which should be printed as 0. The string in the input ends with `EOF`. *Warning: Number values may be greater than the maximum value of type `int` or any primitive type in C.* Hint: the `int getchar()` function can be used to read a character.

**IPA #15: Lab03 - Ex05.** Write a program that reads a sequence of messages and prints them out, one per line. Each message is delimited by quotation marks (character `"`). The message can contain an "escape sequence" - the character loses special meaning if it is preceded by the character \ (backslash). For example, the input \“af"oobar\” matches the message \‘a’fobar\’. So the backslash allows you to include quotes in the message just like the backslash itself.

**IPA #16: Lab03 - Ex06.** Write a program that reads a positive integer from the input (such as a sequence of characters up to 100 chars) and that decides whether the number read is divisible by 9. If the number is divisible by 9, the program should print the message 'yes', and should print 'no' otherwise. *Warning: Number values can be greater than the maximum value of type 'int' or any primitive type in C.* Hint: A number is divisible by 9 iff the sum of its digits is divisible by 9. For example, the sum of the digits of the number 729 is 18, so it is divisible by 9. The fact can be seen from the following equation: \(7 \times 100 + 2 \times 10 + 9 = (7 \times 99 + 7) + (2 \times 9 + 2) + 9\).

**IPA #17: Lab03 - Ex07.** Write a program that takes a sequence of numbers and operators \(+/\cdot\) representing an arithmetic expression and returns the result of that arithmetic expression. The string in the input ends with \'n\'. You can assume that every two numbers are always separated by \'space, operator, space\', i.e., \'op\', for either of the 2 operators above. Example: Input \(70 + 22 - 3\) should return \'89\'. Hint: You should start by converting a sequence of digits (characters) to an integer.

**B.3 Lab04 - Vectors and Strings.**

In this lab, the students learn how to program with integers arrays and strings.

**IPA #18: Lab04 - Ex01.** Write a program that asks the user for a positive integer \(n < \text{VECMAX}\), where \(\text{VECMAX}=100\). Then read \(n\) positive integers. At the end the program should write a graphical representation of the values read as follows. The graph shown is the example for \(n = 3\) and values \(1\ 3\ 4\).

**IPA #19: Lab04 - Ex02.** Write a program that asks the user for a positive integer \(n < \text{VECMAX}\), where \(\text{VECMAX}=100\). Then read \(n\) positive integers. At the end the program should write a graphical representation of the values read as follows. The graph shown is the example for \(n = 3\) and values \(1\ 3\ 4\).

**IPA #20: Lab04 - Ex03.** Write a program that asks the user for a positive integer \(n < \text{VECMAX}\), where \(\text{VECMAX}=100\). Then read \(n\) positive integers. At the end the program should write a graphical representation of the values read as follows. The graph shown is the example for \(n = 3\) and values \(1\ 3\ 4\).

Consider that in the following IPAs, all strings have a maximum of 'MAX = 80' characters (including the end-of-string character).

**IPA #21: Lab04 - Ex04.** Write a program that reads a word from the terminal and checks whether the word is a palindrome or not. A word is a palindrome if it is spelled the same way from left to right and vice versa (eg "AMA" is a palindrome). If the word is a palindrome, the program should print the value 'yes', and 'no' if not. *Hint: You can use \‘scanf\(%s\', s\) to read a word. Note that the string \‘s\' does not ask for \‘&\' in \‘scanf\'.

**IPA #22: Lab04 - Ex05.** Write a program that reads characters from the keyboard, character by character, until it finds the character \‘ñ\' or EOF and writes the line read to the terminal. Implement the \‘int leLinha(char s[])\’ function which reads the line into the string \‘s\' and returns the number of characters read. *Hint: After solving this exercise, try using the \‘fgets\' command.

**IPA #23: Lab04 - Ex06.** Write a program that reads a line from the terminal (use the function from the previous exercise) and writes the same text to the terminal, but with the lowercase letters replaced by the respective uppercase letters. Implement the \‘void upperCase(char s[])\’ function. *Note: Remember that the string \‘s\’ is changed by the ‘uppercase’ function.

**IPA #24: Lab04 - Ex07.** Write a program that reads a line and a character and writes to the terminal the same line where all occurrences of the character were removed. Implement the \‘void eraseCharacter(char s[], char c)\’ function that erases the character \‘c\’ from the string \‘s\’.

**IPA #25: Lab04 - Ex08.** Write a program that reads two integers in decimal representation and prints the larger of those two numbers. You can assume that the two numbers have the same number of digits and a maximum of 100 characters. *Note: The numbers may be too large to be stored in a \‘long long\’ variable, for example \‘998888888888888888887\’ and \‘9988888888888888888888\’.\n
---

IPA #19: Lab04 - Ex02. Write a program that asks the user for a positive integer \(n < \text{VECMAX}\), where \(\text{VECMAX}=100\). Then read \(n\) positive integers. At the end the program should write a graphical representation of the values read as follows. The graph shown is the example for \(n = 3\) and values \(1\ 3\ 4\).