February 2018

CS2303 AutoGrader

Jonathan Henry Morse
Worcester Polytechnic Institute

Follow this and additional works at: https://digitalcommons.wpi.edu/mqp-all

Repository Citation

This Unrestricted is brought to you for free and open access by the Major Qualifying Projects at Digital WPI. It has been accepted for inclusion in Major Qualifying Projects (All Years) by an authorized administrator of Digital WPI. For more information, please contact digitalwpi@wpi.edu.
CS2303 AutoGrader

A Major Qualifying Project Report submitted to the faculty of
Worcester Polytechnic Institute
In partial fulfillment of the requirements for the degree of Bachelor of Science
by:

Jonathan Morse (Computer Science)
February 27, 2018

Submitted to:
Professor Hugh C. Lauer, WPI Advisor

This report represents the work of WPI undergraduate students submitted to the faculty as evidence of completion of a degree requirement. WPI routinely publishes these reports on its website without editorial or peer review. For more information about the projects program at WPI, please see http://www.wpi.edu/academics/ugradstudies/project-learning.html
Abstract

The number of undergraduate computer science students at Worcester Polytechnic Institute has increased in recent years. Due to the increase, strain has been put on teaching assistants for grading programming assignments. In order to lessen this strain we developed a python-based autograder for the programming assignments in CS2303. These auto graders semi-automate the grading process which allows teaching assistants to spend more time helping students in office hours and less time grading.
<table>
<thead>
<tr>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2.1</td>
</tr>
<tr>
<td>2.2</td>
</tr>
<tr>
<td>2.3</td>
</tr>
<tr>
<td>2.3.1</td>
</tr>
<tr>
<td>2.3.2</td>
</tr>
<tr>
<td>2.3.3</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>3.1</td>
</tr>
<tr>
<td>3.2</td>
</tr>
<tr>
<td>3.3</td>
</tr>
<tr>
<td>3.4</td>
</tr>
<tr>
<td>3.5</td>
</tr>
<tr>
<td>3.6</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>4.1</td>
</tr>
<tr>
<td>4.1.1</td>
</tr>
<tr>
<td>4.1.2</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>A.</td>
</tr>
<tr>
<td>B.</td>
</tr>
<tr>
<td>C.</td>
</tr>
<tr>
<td>D.</td>
</tr>
</tbody>
</table>
1 Introduction

In recent years, the number of students in each new class of undergraduate students at WPI is continuously setting the record for largest class in history. This increase in students is putting a significant strain on the computer science department. With the influx of students, grading the programming assignments of introductory computer science courses is becoming a time consuming task. More teaching assistants than ever before are required, and even with that increase, less time is actually being spent helping students learn. Teaching assistants grade quizzes, tests, and assignments with professors to allow grades to be returned in a timely manner for large classes. They also host office hours were a student can come and ask a teaching assistant any question about the course. Recently, teaching assistants are spending significantly more hours grading work than hosting office hours. Ideally, most of a teaching assistant’s time would be spent in office hours working with students one on one, to ensure students grasp the concepts of the introductory courses. The goal of this project is to research and develop a way to decrease the number of hours a teaching assistant needs to spend grading in CS2303: Introduction to Systems Programming. This introductory class is usually taken by a student during C-Term of freshman year. It is most likely taken after a student has completed two previous programming classes. It introduces the C and C++ programming languages, which are harder to understand than the languages used in previous classes.

The best way we thought of accomplishing this goal was by automating as much grading of programming assignments as possible. We had to research and develop an autograder, or a program that grades another program. We researched autogrders that were available and used in general and at the university level. To find general purpose autogrders we searched the internet, and to find autogrders used at other universities we emailed professors directly. We were hoping to find an
autograder with three characteristics, namely to integrate well with existing programming assignments, to be simple to maintain and, to work reliably.

In response to professors we got valuable input from Professor Armando Fox [1] a computer science professor at the University of California at Berkeley. He had previously created an autograder from scratch for one of his classes, so he had experience. What he mentioned was, if we were unable to find an existing autograder that fit our needs and had to make one from scratch, to aim to autograde 80% of each assignment. In his experience it was much easier to make an autograder that graded 80% of an assignment, than one that grades 100% of each assignment. Since the autograders are going to be used by teaching assistants and they have the ability to grade programming assignments along with the autograders, we thought 60% to 80% automation would accomplish the goals of the project. We specifically focused on automating redundant task like organizing files and compiling code, because those are tasks machines do more efficiently than humans.

After we conducted research, the best autograder we found to suite the goals of the project was AutoLab [2]. AutoLab is under active developed by a team at Carnegie Mellon University led by Professor David O’Hallaron [3]. It is a system that facilitates in autograding programming assignments. After attempting to use AutoLab and working with it to create our autograders, we determined it was not suited well to the goals of the project. The suite of features that AutoLab provided were mostly unnecessary, for the scope of this project. Also, we believe that AutoLab would not be simple and reliable when being used by the teaching assistants.

We then transitioned to creating an autograder from scratch. This allowed us to create an autograder that was specifically designed for CS2303’s assignments and to have a complete understanding of how the system works. We were able to develop a series of programs written in the Python programming language, to partially grade some of the programming assignments in CS2303.
We first started by creating a basic user interface that was simple to use, then began developing functions to unzip directories and compile student code. Lastly, we were able to create functions that could test the output of student’s code for correctness.

2 Initial Research

We divided the research portion of the project into two phases, first we conducted Internet research to investigate autograder use in general. Then, we emailed college professors directly to determine autograder use at the university level.

2.1 Autograders in General

Our initial research started on the Internet, in our searches we were able to find two particularly useful links. One was from researchgate.com [4] and the other was from stackexchange.com [5]. Both links were from reputable websites, and the commenters seemed to be either computer science students or computer science professors. Information in those links introduced us to two useful topics related to autograders.

First was MOOCs, which stands for Massive Open Online Courses. MOOCs are taken by thousands or millions of people online, and those students are taught through recorded presentations, interactive labs and programming assignments. Due to the size of these courses, feedback for labs and assignments needs to be provided without human intervention. Therefore, they make extensive use of autogrders.

One website that offers MOOCs is Stepik.com [6]. Stepik provides premade course but also allows users to develop their own courses that can be public or private. Most of the teaching is done by explaining a topic in a few paragraphs of text, then having a student develop one or two lines of code at a time, this step-by-step process slowly turns those chunks of code into a program. The programming assignments for CS2303 are open-ended. Students produce entire programs from
scratch before submitting for grading. For this reason, research or websites that focus on MOOCs were avoided for the remainder of the project.

Second, we learned about four autograders from our internet searches. Those were Junit [7], Web-Cat [8], Mooshak [9] and AutoGradr [10].

- Junit is a framework for testing functions written in the Java programming language. It was originally developed by researchers out of the University of Calgary. Junit is very popular and is one of the most widely used external libraries in Java [11].

- Web-Cat is a system that determines grades based on the test cases the students make, not those made by the professor or teaching assistant. According to web-cat.org, it is a web application with a plug-in-style architecture.

- Mooshak is a code testing system originally used for keeping score of programming competitions. Due to its popularity it was then turned into a product for classroom use. The main developer is Professor José Paulo Leal out of the University of Porto [9].

- AutoGradr is a website were users can create courses, consisting of labs and projects. Once a course is created, AutoGradr provides tools to autograde the labs and projects within a web browser. According to AutoGradr.com, it “handles collecting student code, compiling and running them against test cases, providing instant feedback to students”.

The only product that we determined could be used for this project was AutoGradr. All the other products either did not support C and C++ or were old and not under active development.

2.2 Autograders at the University Level

To get information on autograders used at other universities we emailed the computer science department heads of twenty-five of the highest ranking undergraduate computer science programs in the nation. Of the twenty-five schools, we got no response from sixteen. Four out of
the remaining nine schools either made their own autograder or did not use one. Of the remaining five schools, the recommended autogrades were ZyBooks [12], EdX [13], GradeScope [14] and AutoLab [2].

- ZyBooks is a learning platform for programming languages that are accessible via a web browser.
- EdX is a collaboration between many high ranking universities including MIT and Harvard to create a collection of free, college-level MOOCs in many subjects.
- GradeScope is a website used to speed up the grading of exams and quizzes that have repeatable answers, which also has an autograding tool for programming assignments. GradeScope was initially developed by Professor Peter Abbeel [15] of the University of California at Berkeley.
- AutoLab is an open source application that is being developed by a team at Carnegie Mellon University led by Professor David O’Hallaron. It provides a web interface to submit work, and view grades, which compliments a back-end auto grader written in Python, it also has a supplemental website that provides information on how to interact with the application.

AutoLab and GradeScope appeared to suit our needs so we investigated them further online. GradeScope unfortunately charges a fee to use the website and is not open source. We wanted something that was free and open source to avoid having to secure funds for this project and so that we could make changes to the product if it was in our own best interest. AutoLab appeared to be similar to GradeScope but was free and open source, so we decided to give it a try.

2.3 AutoLab

AutoLab is being actively developed by a team of undergraduate and graduate students at Carnegie Mellon University. Currently, a local version of AutoLab can be downloaded from autolabproject.com, this local version is used at CMU by over twelve hundred students each
semester. However, the AutoLab team is working on a cloud-based system that would allow anybody in the world to use AutoLab via the web.

2.3.1 Installation

We installed the local version of AutoLab. The AutoLab website provides installation instructions and what the website calls the “one-click install” option. This option only worked on the Ubuntu operating system [16] so we created an Ubuntu virtual machine using VirtualBox [17]. This option makes use of GitHub.com [18], a website that allows users to host source code that uses the Git [19] version control system to manage and to track the changes made to files. All we had to do was clone the required files from GitHub and run a shell script within the Ubuntu virtual machine. This simple installation process is possible because the shell script installs containers [20].

Containers allow a software application to be installed with all files and other dependencies it needs to run correctly, with no work from the end user. The developers at CMU used Docker [21] to create these containers, Docker is an application that provides an API to create and work with containers. The shell script installed three separate containers. The first container contained all the code needed to run the web interface, the second was a SQL database that stored all the information and data being shown on the web interface, lastly, the third container was called Tango and contains all the code that allows the system to autograde files. Even though the installation of AutoLab is supposed to be simple, we did receive errors during installation and had to reinstall multiple times.

2.3.2 Configuration

We were able to create a course and an assignment within AutoLab once we got it running. We initially used the sample assignment that the AutoLab developers created called Hello Lab. We downloaded the hello.tar file from AutoLab’s GitHub [22] page and were able to install that file into AutoLab using the web interface. The way the web interface works is, each student, teaching assistant, and professor has an account for AutoLab, which provides different levels of access and
responsibility. A student for example can log in, select a specific course and submit a file to a specific assignment within the course, submitting files is the only action student accounts are allowed to do. This file would be saved by AutoLab into a specific folder and could be downloaded by the teaching assistant or professor. Unfortunately, the first time we attempted to submit a file to the Hello Lab assignment we got an error message. The error message was vague, all it stated was that the autograder was unable to grade the file, with no mention as to why. This was unusual because we made no changes to AutoLab and the lab that was used was made by the developers.

Next we attempted to create our own assignment and not use the hello.tar file assuming that was causing the problem. The web interface of AutoLab is nicely organized and visually appealing, we used the web interface to create our custom assignment, which was a simple process. However, we also had to write code that could interact with the web interface to autograde student submissions, AutoLab does this using JSON objects [23]. JSON stands for JavaScript Object Notation, and it allows for easy translation of data between a software application and a web server.

We were able to create a very simple program that returned JSON objects, but ran into serious difficulty integrating the program with AutoLab. This process involves creating a series of directories and makefiles and installing them within AutoLab. Specifics about all the makefiles and directories can be found at https://autolab.github.io/docs/lab/. Even though the website provides detailed information on what needs to be in the makefiles and directories, when we attempted to this it did not work properly. It was difficult to figure out what exactly was going wrong, and what we needed to fix. AutoLab provided no error or warnings messages that could give us hints into the cause of our problems.

We also struggled with Tango, another part of AutoLab. Tango is the application that interacts with user created programs using JSON objects. We developed our program based on information from the AutoLab website, and ensured that our program properly returned JSON
object. However, we still could not get AutoLab to grade any file we submitted. Unfortunately, the information about Tango on the AutoLab website is only useful if Tango is installed as a standalone application. When using the one click install, Tango should work without any setup. So, when we had trouble with Tango we were unsure what we had to do correct things, and what we were doing wrong.

2.3.3 Concerns

The main issue we had with AutoLab was determining the reason AutoLab would not grade a file we submitted. We could not determine if the problems were stemming from a misconfiguration of Tango, improper use of JSON objects in our program, or missing directories and makefiles. This did not allow us to properly investigate AutoLab, and determine its effectiveness at accomplishing the goals of the project.

Another problem with AutoLab is it requires a web server and user accounts. When we were testing out AutoLab we installed the “local” version. This version provides a dummy account and is used to just learn how works. If we wanted to deploy AutoLab for class room use we would need to install the “deployment” version. This version needs the ability to be accessed via a web address, which would require us to create and maintain a web server. Also the deployment version of AutoLab requires an email server. The email server is used to create and maintain user accounts. If students ever had difficulty accessing their account or resetting a password, it would cost teaching assistants a few hours of work each term. Due to one of our design goals being simplicity, the need for all these additional systems was a major negative. Although, we learned many valuable things from this experience we decided to no longer use AutoLab due to the many problems described above we decided no longer use AutoLab. The point of this project was to decrease time the teaching assistants have to spend on grading. The best way we believe that could be accomplished is by creating a simple to use autograder that is reliable. AutoLab is a complex piece of software that
has many points of failure and involves many systems working together correctly. We decided to develop our own autograder with a focus on simplicity. We wanted a solution that did not require user accounts, web servers, or emails.

3 Custom Built CS2303 AutoGrader

The point of this project was to decrease time the teaching assistants have to spend on grading. The best way we believe that could be accomplished is by creating a simple to use autograder that is reliable. AutoLab is a complex piece of software that has many points of failure and involves many systems working together correctly. We decided to develop our own autograder so we could have understanding of how the entire system works and to make the focus on simplicity,

3.1 Overview

(See Appendix C for information on downloading and using CS2303_Autograder)

Our autograder is a command-line application written in Python and is invoked using the operating system’s shell. Below is a high-level step-by-step process of how the system works:

Step 1: User inserts zip file into Insert_Zip_File_Here directory (see section 4.1.1.1 for more information)

- This zip file should contain individual student projects, which should also be zip files.

Step 2: User invokes CS2303 Autograder from command line

Step 3: User answers the questions that the autograder asks which include first name, last name and the programming assignment user would like graded
Step 4: Autograder unzips zip file and creates a list of each student who’s project is in the zip file, then unzips each individual student’s zip file

Step 5: Autograder moves a student’s folder into the specified programming assignment folder

Step 6: Autograder invokes specified programming assignment’s grading file, that file grades project, outputs a rubric and then deletes the project from the directory

Step 7: Steps 5 and 6 are repeated until each student’s project is graded

3.2 Getting Started

Before we started to build our autograder we wanted to understand what needs to be graded, to do this we completed the first programming assignment for CS2303. The first assignment is to design a program that accepts specific year as input and outputs the twelve month calendar of that year. The main difficulty of the program is accounting for leap years. A leap year occurs every four years, except every one-hundred years, except every four-hundred years. So the year 2000 was a leap year but the 1900 was not, for example. After finishing the first assignment we believed that an autograder would need to do many string manipulations and interact with the shell a large amount. We had experience coding in the Python programming language, and believe Python has functions that allow easy interaction with the shell and simple string manipulation. Python is also a readable language, which would be beneficial if the autograder needs to be changed or built upon in the future.

3.3 Programming Assignment One

Our first iteration of the autograder for Programming Assignment One was monolithic, the only custom made function was the main function and was hard to follow. It was obvious to us that changes needed to be made. The aspects of the program that would most likely be reused when writing the autograder for the other assignments were compiling the code and printing a rubric. That is when we created autograder_core.py, this file was meant to contain all custom-built functions for
all grader files so that each individual file could pull from the core file and use the same function. This made the grader files look significantly neater and made it is easier to make changes to the code. With this system, if we wanted to make a change, we could change the function in the core file and it would change it for every grader file. We found that debugging the code and understanding the entire scope of the autograder was much simpler with this system.

We first built functions for compiling code and creating a rubric, we then took what we did in the grader file for the first assignment and turned those actions into a series of functions. The first grader was broken up into four functions, were one function built off the previous function. We believed this would make it easier to reuse functions later.

We realized when developing the first grader that it was easiest to grade the output of a program. For the first three programming assignments no function names are mandatory, and no function stubs or pre-written program shells are given to students. The student starts with a blank file and must create the entire program from scratch. The professor that teaches this class believes this is the most effective way for students to learn how to code in C, so we believed it was best to work around this design constraint. So what we thought we would do is standardize the output (see Appendix D for details on the standardized outputs of the first three programming assignments).

We also attempted to create autogrpaders that allowed teaching assistants to never have the need to compile and run the program. We believed that computers more efficiently compile, run and test code than humans can. However, teaching assistants can do a far superior job when it comes to checking a programs style and checking aspects of a program that cannot be tested based on the output. We did not attempt to autograde any style aspect of a program but did try to ensure that everything that involved compiling or running a student’s program was autograded.

Every programming assignment has a certain number of tasks that a student must accomplish to earn full points on the assignment. These tasks are how the professor and teaching
assistants can measure how well a student implemented the design goals of a programming assignment. For the first programming assignment there are seven tasks, the autograder is able to grade the first four tasks

- **Task 1: Correct compilation without warnings**
  - This task is autograded using the `compile_test` function (see Appendix A.2). This function is able to run the command used to compile the project. It checks if the compile was successful then returns the points the student earned and the reason the student earned that score.

- **Task 2: Correct execution with graders’ test cases**
  - This task is autograded using the `check_year` function (see Appendix A.8). This function compares the output of the student’s code for a specific year to a pre-written project that to our knowledge is always correct. It then returns 0 if the outputs were different or 1 if outputs were the same. The result of the function is then used to determine the students score and reason for score.

- **Task 3: Correct usage of scanf() to get inputs from user**
  - This task is autograded using the `search_words` function (see Appendix A.6). This function ensures that the first month of the year and the inputted year is printed in the output in the correct spot. This shows that scanf() was implemented correctly because year that was inputted by the user was printed out in the output, showing the program successfully accepts user input.

- **Task 4: Correct usage of print() to print the various lines of the calendar**
  - This task is autograded using the `check_month` function (see Appendix A.7). This function ensures that the student properly lined up the dates in a certain month with the days of the week, with correct formatting.
• Task 5: Correct usage of conditional and loop statements
  
  o This task is autograded using the `check_year` function again (see Appendix A.8). It checks to ensure the year 2000 prints out correctly. We used this year because it requires that all the leap year calculations are correct, making it one of the hardest years to correctly print out.

  The remaining tasks we were not able to autograde. They involve manual inspection, and the correctness cannot be determined by the output of the student’s code. To combat this problem we expanded on the task descriptions within the programming assignment description to give teaching assistants a better understanding of what to look for. We believe this will reduce the time it takes to grade these tasks significantly. Two aspects of grading that take time is for a teaching assistant to determine what exact to look for in students code, and to answer students emails or questions asking why she received a certain score on a task. With specific instructions on what to look for, teaching assistants will not need to think as much when grading, and will be able to be transparent as well as direct with students, which will reduce follow up questions and in person meetings. The remaining three tasks should be graded as follows.

  • Task 6: Satisfactory README file – 2 Points
    
    o Explanation of how a student’s program works - .5 Points
    o Explanation of how to run program - .5 Points
    o Includes information about use of outside sources – 1 Point
      
      ▪ If a student did use an outside source, source must be cited with at least a one sentence explanation per source.

  • Task 7: Loop invariant for each loop in comments in the code and also in the README document
3.4 User Interface

Next, while running the first autograder many times, we realized it would be annoying for a teaching assistant to have to find the specific grader file she wants to use. To combat this problem we created a file called `CS2303_autograder.py` (see Appendix B), which when ran displays a simple to use menu within the command line. Then the program asks for the teaching assistants first and last name, as well as the programming assignment she would like graded. Figure One below shows what the autograder interface displays.

![Figure One: Autograder interface](image)

This interface allows the teaching assistant to see visually that the autograder is running and to use the same command to grade any assignment.

Once we finished the interface we were worried about how the teaching assistant would give the autograder the file or folder she needs graded. Our solution was to create a simple to understand
structure so everything was organized and have a folder that was called “Insert Zip File Here” (see section 4.1.1.1), within that structure. A teaching assistant would insert the zip file that contained all of the student’s code into that folder. Then we created a function called unzip_organize (see section A.1) that is used within CS2303_Autograder.py, this function that’s unzips each zip file and the records the WPI username of the student based on the file name. Then CS2303_Autograder.py moves student folders one at a time into whatever programming assignment directory the teaching assistant chose (see section 4 for information on files and directories).

Upon further testing, another problem occurred when student’s code did not properly compile. When that occurred, it would cause the autograder to return an error because no file was available. Also in the program description it states that students lose points if their code does not properly compile. So we created a function called compile_test (see Appendix A.2) that would produce a list of student names whose code did not compile. This list allows the program to continue running and grade other student’s code as well as gives the teaching assistants a list of students with incorrect code, so they simply can email the students, inform them of the points lost and ask them to resubmit the project.

3.5 Programming Assignment Two

We believed we had worked out the initial problems with the first autograder enough to begin completing the next assignment. The major difference between this assignment and the first assignment from our perspective was the use of the “make” command and student submissions containing multiple files. We combatted these problems by developing a function that tested if a student has a functioning makefile, and if a student’s program properly compiles using the “make” command. Also, we created a function that ensured that a student ran the “make clean” function before submitting the project, which was something that needed to be done according to the program assignment description.
In Programming Assignment Two a student must create a program that implements the Game of Life. The Game of Life was created in the early 1970s by John Conway, a detailed description of the game can be found at [http://web.stanford.edu/~cdebs/GameOfLife/](http://web.stanford.edu/~cdebs/GameOfLife/). This assignment has ten tasks, each worth four points for a total of forty points. The autograder we developed was able to grade eight out of the ten tasks.

• Task 1: Correct makefile to build program and individual components and to clean up
  
  o We created a function called `make_all_files_test` (see Appendix A.9) that is able create a list of all the “.c” files a student created then attempt to compile each of those individual files, once it does that it attempts to compile the entire project and then clean the entire project. Using the results of the function we were able to assign a student a score and a reason for that score.

• Task 2: Correct compilation without warnings (using –Wall)
  
  o We used the `compile_test` (see Appendix A.2) function again to grade this task. It runs the “make” command then returns a score and a reason for that score.

• Task 3: Correctly reading the initial configuration and centering it in the array
  
  o We created a function called `array_compare` (see Appendix A.11) that allowed us to compare a correct output of the Game of Life to the students output. We were able to test if a student correctly centered and configured the array using this function.

• Task 4: Correct allocation of arrays at run time – 4 Points
  
  o We used the `array_compare` (see Appendix A.11) function to test for this task. We ensured that the students code could produce arrays of different sizes, which ensures that arrays are created at run time.

• Task 5: Correct use of two-dimensional arrays – 4 Points
• Task 6: Correct implementation of game function
  o Using the array_compare (see Appendix A.11) function again, we were able to check if a student properly determined if an occupied cell survived or died and if an unoccupied cell gave birth.

• Task 7: Correct test for termination
  o We created a function called termination_condition (see Appendix A.12) that was able to ensure that a student properly ended the game when one of the four termination conditions were met.

• Task 9: Correct execution with graders’ test cases
  o Using array_compare (see Appendix A.11) we checked if the output from student’s code was correct based on our test cases.

The remaining tasks we were not able to autograde and just like the first assignment, we expanded on the task descriptions within the programming assignment description to give teaching assistants a better understanding of what to look for. The remaining two tasks should be graded as follows:

• Task 8: Satisfactory test cases – 4 Points
  o Output of each test case is contained within submission – 1 Point
  o Explanation of each test case is in README document – 1 Point
  o Test case(s) are non-trivial and show that the student’s program properly implemented the Game of Life – 2 Points

• Task 10: Satisfactory README file, including loop invariants – 4 Points
o Explanation of how a student’s program works - .5 Points

o Explanation of how to run program and any problems student had - .5 Points

o Includes information about use of outside sources – 1 Point
  ▪ If a student did use an outside source, source must be cited with at least a
    one sentence explanation per source.

o One or two sentences explaining each loop invariant – 1 Point

o First line of each loop invariant explanation includes the file name and line number
  related to the specific loop invariant – 1 Point

3.6  **Programming Assignment Three**

For the third assignment students must create a program that scans one or more input text
files. Then use a binary tree to record all of the words in the file, then output a text file, that lists all
the unique words and the number of times they occurred. The assignment is graded based on eight
tasks that are all worth five points each, and has a ten point extra credit task. The autograder we
developed was able to grade six out of the eight tasks.

- Task 1: Correctly build from the makefile without warnings (with –Wall switch)
  o We used the `compile_test` (see Appendix A.2) function again to grade this task.
    It runs the “make” command then returns a score and a reason for that score.

- Task 2: Organization of program into at least three modules
  o We created a function called `counting_modules` (see Appendix A.13) that
    ensures that the students program is made up of at least three or more “.c” files, then
    returns the score and the reason for the score.

- Task 3: Correct construction of binary tree and insertion of nodes
- We created a function called `total_word_scan` (see Appendix A.15) that ensures the output file properly lists the number of unique words and the total number of words.

- **Task 4: Correct traversal of binary tree and output of information according to specified format**
  - We created a function called `individual_wordScan` (see Appendix A.16) that ensures that the format of the output file is correct, contains the proper number of words, and proper count of those words.

- **Task 5: Correct use of malloc() and correctly freeing all malloc’ed data – 5 Points**
  - We used the function `individual_wordScan` (see Appendix A.16) to ensure that the code did not produce a segmentation fault, to ensure malloc() was used correctly.

- **Task 7: Correct execution with graders’ test cases**
  - We combined `total_word_scan` (see Appendix A.15) and `individual_wordScan` (see Appendix A.16) to ensure the output file is correct for our test cases.

The remaining tasks we were not able to autograde, we expanded on the task descriptions within the programming assignment description to give teaching assistants a better understanding of what to look for. The remaining two tasks should be graded as follows:

- **Task 6: Proper destruction of tree and all of its objects before exiting – 5 Points**
  - Using the free() command to deconstruct each object – 2.5 Points
  - Using the free() command to deconstruct tree – 2.5 Points

- **Task 8: Satisfactory README file, including output of two non-trivial test cases – 5 Points**
○ Explanation of how a student’s program works - .5 Points

○ Explanation of how to run program and any problems student had - .5 Points

○ Includes information about use of outside sources – 1 Point
  - If a student did use an outside source, source must be cited with at least a one sentence explanation per source.

○ One or two sentences explaining each loop invariant with first line of each loop invariant explanation including the file name and line number related to the specific loop invariant – 1 Point

○ At least two test cases, with the output in the submission and an explanation of each test case in the README document – 2 Points

4 Description of Directories and Files

For the below subsections, all files or directories that are within the same section level (ex. Section 4.1 and Section 4.2 are on the same level) are within the same directory level. With each subsection is one level lower than its parent section in the directory structure.

4.1 CS2303_AutoGrader

4.1.1 src

4.1.1.1 Insert_Zip_File_Here

This directory is where the user will input the zip file that contains each individual student’s project, each student project should also be a zip file.

4.1.1.2 Output_Files

4.1.1.2.1 Rubrics

This directory is where the rubrics for each student will be after the autograder is run.
### 4.1.1.2 not_clean.txt

This file will contain the name of any student who did not run the “make clean” command before submitting the project. The students who are on this list will lose points on the project. This file is deleted when running the autograder for programming assignment one because it is not needed.

### 4.1.1.2.3 did_not_compile.txt

This file will contain the name of any student whose code failed to compile. This list can be used by the teaching assistants to email those specific students to resubmit working code.

### 4.1.1.3 PA1

#### 4.1.1.3.1 PA1_autograder.py

This file uses functions from autograder_core.py and calls to the shell to compile, grade and create a rubric for the first programming assignment.

#### 4.1.1.3.2 PA1_Test.c

This file is a correctly implemented program for the first assignment. This file is used in the check_year function found in autograder_core.py. It is used to test the output of a student's program to the correct output. In our testing we have found no problems in this file.

### 4.1.1.4 PA2

#### 4.1.1.4.1 PA2_autograder.py

This file uses functions from autograder_core.py and calls to the shell to compile, grade and create a rubric for the second programming assignment.
4.1.1.4.2 task_*_test_*_answer.txt

These files are used in the array_compare and termination_condition functions found in autograder_core.py. These functions compare this text file to the output of a student's program.

4.1.1.4.3 task_*_test_*_input.txt

These files are used in the array_compare and termination_condition functions found in autograder_core.py. These text files are needed to run programming assignment two.

4.1.1.5 PA3

4.1.1.5.1 PA3_autograder.py

This file uses functions from autograder_core.py and calls to the shell to compile, grade and create a rubric for the third programming assignment.

4.1.1.5.2 individual_word_test_answer.txt

Used as one of the arguments in individual_word_scan function in PA3_autograder.py. Contains all the individual words that should be found in the input files.

4.1.1.5.3 program_test_input_one.txt

Used as one of the arguments in both the individual_word_scan and total_word_scan functions in PA3_autograder.py. It is a text file of the Martin Luther King Jr. speech.

4.1.1.5.4 program_test_input_two.txt

Used as one of the arguments in both the individual_word_scan and total_word_scan functions in PA3_autograder.py. It is a text file of a Roosevelt speech.
4.1.5.5 total_word_test_answer.txt

Used as one of the arguments in total_word_scan function in PA3_autograder.py. Contains information about total number of words.

4.1.6 PA4

4.1.6.1 PA4_autograder.py

This file is a template for the programming assignment four autograder. It uses functions from autograder_core.py to print the rubric for programming assignment four.

4.1.7 PA5

4.1.7.1 PA5_autograder.py

This file is a template for the programming assignment five autograder. It uses functions from autograder_core.py to print the rubric for programming assignment five.

4.1.8 PA6

4.1.8.1 PA6_autograder.py

This file is a template for the programming assignment four autograder. It uses functions from autograder_core.py to print the rubric for programming assignment four.

4.1.9 autograder_core.py

This file contains all custom made functions used to unzip files, compile code, grade projects and create rubrics.
4.1.1.10  CS2303_autograder.py

This file provides an interface for the user, then unzips student projects using the unzip_organize function in autograder_core.py. Then it moves student projects to the specified autograder and invokes the autograder.

4.1.2  README.txt

This file provides information about the autograder. It contains general information, instructions on how to run the autograder, a description of files and directories and information about each argument for each custom made function in autograder_core.py.

5  Future Work

Currently, we believe that our system makes it unnecessary for teaching assistants to compile and run student code for the first three programming assignments. In our opinion, two more crucial steps are needed to make our system complete.

First, an autograder needs to be developed for the last three programming assignments. We believe that custom functions should be developed to grade these assignments, because we have not found any autograders that would be as effective as custom built function. The most efficient way of accomplishing this, in our opinion, is to start with the stub files we have created for the last three programming assignment. Then combine custom built functions with our compile_test, compile_check and clean_check functions. That should be enough to grade 60% to 80% of all three assignments. Second, the autograders need to be tested in the real world. This could be completed the next time CS2303 is offered, by possible having half the teaching assistants use CS2303 autograder and half perform manual grading. Using surveys, observation and interviews gather data on the time requirement, reliability and simplicity of CS2303 autograders compared to manual grading to determine the usefulness of the system.
6 Conclusion

CS2303 AutoGrader provides teaching assistants with a simple, and reliable way of automatically grading most aspects of the first three programming assignments in CS2303. This allows teaching assistants to grade programming assignments much faster than can be accomplished through manual grading, creating more opportunity for teaching assistants to host office hours and help students learn.

7 References


8 Appendices

A. autograder_core.py

We divided up the file into subsections by function, each subsection contains a screenshot of the source code of the function and a description of each of the arguments of the function.

A.1 unzip_organize:

```python
def unzip_organize():
    zip_output_name = os.listdir("./Insert_Zip_File_Here")[0]
    output_name = zip_output_name.split('.')[0]
    subprocess.run("cd ./Insert_Zip_File_Here; unzip %s"
                   % (output_name),
                   shell=True,
                   stdout=subprocess.PIPE,
                   stderr=subprocess.PIPE)

    for x in range(0, number_student_zip_outputs):
        subprocess.run("cd ./Insert_Zip_File_Here/%s; unzip %s; mv %s %s"
                       % (output_name,
                           list_student_outputs[x],
                           list_student_outputs[x],
                           list_students[x]),
                       shell=True,
                       stdout=subprocess.PIPE,
                       stderr=subprocess.PIPE)

    return list_students, output_name
```
A.2 compile_test:

```python
def compile_test(command, grade_possible):
    compile_test_one = subprocess.run(command,
                                       shell=True,
                                       stdout=subprocess.PIPE,
                                       stderr=subprocess.PIPE)

    if compile_test_one.returncode == 0:
        grade_earned = grade_possible
        reason_for_score_for_score = "Compile was successful, no warnings or errors"
    else:
        grade_earned = 0
        reason_for_score_for_score = "Compile failed, program had error(s) or warning(s)"

    return grade_earned, reason_for_score_for_score;
```

**command**: a string, the command used to compile the project (ex. "make")

**grade_possible**: an int, the number of possible points that could be earned for the task
A.3 compile_check:

```python
# Checks the grade earned from the compile test. If it
# is not equal to the highest possible grade then the code
# did not compile. So the function adds the student's name
# to the did_not_compile.txt output.

def compile_check(grade_earned, grade_possible, student_name):
    if grade_earned != grade_possible:
        compile_doc = open("./Output_Files/did_not_compile.txt", "w")
        compile_doc.write("\n%%s \n\n% (student_name))
    else:
        return

return
```

**grade_earned:** an int, compile_check should be run after compile_test, and this argument should be set to the grade_earned that was returned in compile_test  
**grade_possible:** an int, the number of possible points that could be earned for the task  
**student_name:** a string, the name of the student who's code is being graded.

A.4 clean_check:

```python
# Counts the number of files in the students folder then
# runs "make clean", then recounts the number of files.
# If the initial count is greater then adds the students
# name to the not_clean.txt file.

def clean_check(student_name):
    initial_dir_size = len(os.listdir("./%s" % (student_name)))
    clean_command = subprocess.run("make clean",
        shell=True,
        stdout=subprocess.PIPE,
        stderr=subprocess.PIPE)

    cleaned_dir_size = len(os.listdir("./%s" % (student_name)))

    if initial_dir_size > cleaned_dir_size:
        clean_doc = open("./Output_Files/not_clean.txt", "w")
        clean_doc.write("\n% (student_name))
    else:
        return

return
```

**student_name:** a string, the name of the student who's code is being graded.
A.5 test_case:

```python
def test_case(command, program_input):
    run_command = Popen(command, stdin=PIPE, stdout=PIPE)
    output_bytes = run_command.communicate(input=b'%'d' % (program_input))[0]
    output_list = output_bytes.decode("utf8").splitlines()
    return output_list;
```

**command**: a string, should be the command used to run the program

**program_input**: an int, the number that is entered into the program while it is running. This function was designed specifically for programming assignment one.

A.6 search_word:

```python
def search_word(command, program_input, word_one, word_two, grade_possible):
    output = test_case(command, program_input)

    for x in range(0, len(output)):
        if word_one in output[x] and word_two in output[x]:
            grade_earned = grade_possible
            reason_for_score_for_score = "\n\nThe number \%d was entered as user "
            % (program_input)
            + "input, code successfully scanned user "
            + "input and printed it on screen, scanf()"
            + " implemented correctly"
            break
```
**command**: a string, should be the command used to run the program

**program_input**: an int, the number that is entered into the program while it is running. This function was designed specifically for programming assignment one.

**word_one**: a string, should be a month of the year (ex. "June")

**word_two**: a string, should be the string form of program_input. (ex. if program_input = 20, then word_two = "20")

**grade_possible**: an int, the number of possible points that could be earned for the task

---

A.7 check_month:

```python
def check_month(command, program_input, char1, char2, month, grade_possible):
    output = test_case(command, program_input)

    # Initializes a list that will contain all strings of a particular month
    month_list = []

    for x in range(0, len(output)):
        if month in output[x]:
            month_list.append(output[x + 1])
            month_list.append(output[x + 2])
            break
        else:
            result = 0
            continue
```
command: a string, should be the command used to run the program

program_input: an int, the number that is entered into the program while it is running. This function was designed specifically for programming assignment one.

char_one: a char, should be the last letter of one of abbreviated days of the week. (ex. The abbreviation for Wednesday is Wed so char_one should equal 'd')

char_two: a char, should be a number anywhere from 1 to 7 in string form. (ex. '4')

month: a string, should be the name of a month (ex. "June")

grade_possible: an int, the number of possible points that could be earned for the task

A.8 check_year:

```python
# Error handling
if char1 in month_list[0] and char2 in month_list[1]:
    if month_list[0].index(char1) == month_list[1].index(char2):
        grade_earned = grade_possible
        reason_for_score_for_score = "\n\n\tCorrectly printed first week of" \n+ " calendar for year %d, printf() \n% (program_input) \n+ " implemented correctly"
    else:
        grade_earned = 0
        reason_for_score = "\n\n\tIncorrectly printed first week of" \n+ " calendar for year %d, printf() \n% (program_input) \n+ " not implemented correctly"
else:
    grade_earned = 0
    reason_for_score = "\n\n\tCorrectly printed first week of" \n+ " calendar for year %d, printf() \n% (program_input) \n+ " implemented correctly"

return grade_earned, reason_for_score
```
# A list of all the months
all_months = ["January",
"February",
"March",
"April",
"May",
"June",
"July",
"August",
"September",
"October",
"November",
"December"]

for x in range(0,len(all_months)):
    for y in range(0,len(test_output)):
        if all_months[x] in test_output[y]:
            first_week.append(test_output[y+2])
            first_week.append(test_output[y+3])
            char_one = first_week[0][32]
            char_two = first_week[1][32]
            month_check = check_month(command,
                                      program_input,
                                      char_one,
                                      char_two,
                                      all_months[x],
                                      1)[0]
            result = month_check + result
            first_week.clear()

if result == 12:
    final_result = 1
else:
    final_result = 0

return final_result

command: a string, should be the command used to run the program
program_input: an int, the number that is entered into the program while it is running. This function was designed specifically for programming assignment one.
A.9 make_all_files_test:

```python
# This function test to see if each individual ".c" file
# compiles, and also test to see if "make" and "make clean"
# were implemented correctly

def make_all_files_test(grade_possible, student_name):
    # Creating a list of all file names in student's folder
    list_of_files = os.listdir("./%s" % (student_name))
    # Number of files in student's folder
    num_of_files = len(list_of_files)
    # Initializing a list that will contain the name of
    # all ".c" files in folder
    code_files = []

    # Finding all the ".c" files
    for x in range(0, num_of_files):
        if ".c" in list_of_files[x]:
            # If a file ends in ".c" the ".c" is removed and
            # ".o" is added then the string is added to the code_files list
            code_files.append(list_of_files[x].split('.')[0] + ".o")
    # Number of source code files
    num_of_code_files = len(code_files)
    # Variables used to calculate if all compile test worked
    total = 0
    # Set to 2 to account for "make" and "make_clean" tests
    tests_run = 2
    tests_score = \\
```
# Goes into the folder and runs make
# for each individual ".o" file
for x in range(0,num_of_code_files):
    output_make_test = compile_test("cd ./%s; make %s"
        % (student_name, code_files[x]), 1)
    total = total + output_make_test[0]
    reason_for_score = reason_for_score \\
        + "\n\t" \\
        + output_make_test[1] \\
        + " when running command: make %s" % (code_files[x])
    tests_run = tests_run + 1

# Runs make on the entire project
make_test = compile_test("cd ./%s; make" % (student_name), 1)
make_clean_test = compile_test("cd ./%s; make clean" % (student_name), 1)
    total = total + make_test[0]
    total = total + make_clean_test[0]
    reason_for_score = reason_for_score \\
        + "\n\t" \\
        + make_test[1] \\
        + " when running command: make "
        + make_clean_test[1] \\
        + " when running command: make clean"

if total == tests_run:
    grade_earned = grade_possible
    return grade_earned, reason_for_score
else:
    grade_earned = 0
    return grade_earned, reason_for_score

**grade_possible**: an int, the number of possible points that could be earned for the task

**student_name**: a string, the name of the student who's code is being graded.
A.10 test_case_two:

```python
# Runs a command that does not need inputs when running,
# unlike test_case and returns stdout as a list of strings.
def test_case_two(command):
    output = subprocess.run(command,
        shell=True,
        stdout=subprocess.PIPE,
        stderr=subprocess.PIPE)

    output_list = output.stdout.decode("utf8").splitlines()

    return output_list
```

**command:** a string, should be the command used to run the program

A.11 array_compare:

```python
# Test to ensure that the expected array that is created
# when running Programming Assignment Two is what is
# actually created. Does this by reading a file that
# contains the correct array and comparing that to the
# array created by the student
def array_compare(command, answer_output, num_of_lines):

    student_output = test_case_two(command)

    # Turn a output into a list line by line
    output = open(answer_output, "r")
    read_output = output.read().splitlines()

    counter = 0

    for x in range(0, num_of_lines):
        if read_output[x] == student_output[x]:
            counter = counter + 1
        else:
            continue

    if counter == num_of_lines:
        result = 1
    else:
        result = 0

    return result
```

38
command: a string, should be the command used to run the program
answer_output: a string, this should be the name of the file that contains the array that should be produced.
num_of_lines: an int, the number of lines that should be recorded of the student program's output so that only the array will be recorded.

A.12 termination_condition:

```python
# This function first ensures that the proper array is produced
# using array compare, then ensures the program stopped due to
# the correct termination condition.
def termination_condition(command, answer_output, num_of_lines, condition):
    # Tests to see if proper array is produced
    counter = array_compare(command, answer_output, num_of_lines)
    # Records entire output of command
    command_output = test_case_two(command)

    for x in range(0, len(command_output)):
        if condition in command_output[x]:
            counter = counter + 1
        else:
            continue

    if counter == 2:
        result = 2
    else:
        result = 0

    return result
```

class: a string, should be the command used to run the program
answer_output: a string, this should be the name of the file that contains the array that should be produced.
num_of_lines: an int, the number of lines that should be recorded of the student program's output so that only the array will be recorded.
condition: a string, the reason why the students program is being terminated

A.13 counting_modules:

```python
# Counts the number of ".c" files in a folder. Then
# returns a grade and a reason for that grade based
# on the number of ".c" files.
def counting_modules(grade_possible, student_name):
    # Creating a list with all files in student's folder
    list_of_files = os.listdir("./%s" % (student_name))
```

39
grade_possible: an int, the number of possible points that could be earned for the task

student_name: a string, the name of the student who's code is being graded.

A.14 word_scan:

```python
def word_scan(command, output_name):
    command_output = test_case_two(command)
    output = open(output_name, "r")
    word_output = output.read().splitlines()
    total_word_output = []
    individual_word_output = []
```
command: a string, should be the command used to run the program
output_name: a string, the name of the file that the students program outputs.

A.15 total_word_scan:

```python
for x in range(0, len(word_output)):
    if "---" in word_output[x]:
        total_word_output.append(word_output[x+1])
        total_word_output.append(word_output[x+2])
        for y in range(0, (x-1)):
            individual_word_output.append(word_output[y])
    else:
        continue

return word_output, total_word_output, individual_word_output
```

command: a string, should be the command used to run the program
output_name: a string, the name of the file that the students program outputs.
test_name: a string, the name of the file that contains the output that will be compared to the students output (ex. "task_four_test_answer.txt")
grade_possible: an int, the number of possible points that could be earned for the task
comment: a string, used to set the reason_for_score variable.
A.16 individual_word_scan:

```python
def individual_word_scan(command, output_name, test_name, grade_possible, comment):
    individual_word_output = word_scan(command, output_name)[2]
    test_output = open(test_name, "r")
    test_output_output = test_output.read().splitlines()

    total_counter = 0
    correct_counter = 0

    for x in range(0, len(individual_word_output)):
        total_counter = total_counter + 1
        if test_output_output[x] == total_word_output[x]:
            correct_counter = correct_counter + 1
        else:
            continue

    if total_counter == correct_counter:
        grade_earned = grade_possible
        reason_for_score_for_score = comment + " was successful"
    else:
        grade_earned = 0
        reason_for_score_for_score = comment + " was unsuccessful"

    return grade_earned, reason_for_score_for_score
```

**command**: a string, should be the command used to run the program

**output_name**: a string, the name of the file that the students program outputs.

**test_name**: a string, the name of the file that contains the output that will be compared to the students output (ex. "task_four_test_answer.txt")

**grade_possible**: an int, the number of possible points that could be earned for the task

**comment**: a string, used to set the reason_for_score variable.
A.17 header:

```python
def header(assignment_number, student_name, grader_first_name, grader_last_name, point_total):
    # Centers the title in a space of 100 total characters
    title = "PROGRAMMING ASSIGNMENT %s RUBRIC" \
            "
            % (assignment_number)
    
    # WPI username of student completing assignment
    student_name = "Student WPI username: %s" \
                    % (student_name)
    
    # Full name of grader
    grader_name = "Assignment graded by: %s %s" \
                   % (grader_first_name, grader_last_name)
    
    # Total points for the programming assignment
    point_statement = "This programming assignment is worth: %s points" \
                       % (point_total)
    
    # Formatting
    header = "%s
            %s
            %s
            %s
            %s" \
             % (title,
                student_name,
                grader_name,
                point_statement)

    return header;
```

**assignment_number**: an string, a string form of a number in all caps. (ex. "TWO")

**student_name**: a string, the name of the student who's code is being graded.

**grader_first_name**: a string, first name of the grader (ex. "Jonathan")

**grader_last_name**: a string, last name of the grader (ex. "Morse")

**point_total**: an int, total number of points for the assignment, (ex. "Forty points(40)")
A.18 task_create:

```python
def task_create(task_number, task_description, points_possible, points_earned, reason_for_score):
    task_statement = "Task %s: %s" \
        % (task_number, task_description)
    points_possible_statement = "Maximum Points: %d" \
        % (points_possible)
    points_earned_statement = "Earned Points: %d" \
        % (points_earned)
    reason_for_score_statement = "reason for score: %s" \
        % (reason_for_score)

    # Formatting
    task = "\n\n%s\n\n%0.3s\n\n%0.3s\n\n%0.3s\n\n%0.3s\n\n%0.3s"
        % (task_statement, \
             points_possible_statement, \
             points_earned_statement, \
             reason_for_score_statement)

    return task;
```

task_number: a string, a number in string form (ex. "one")
task_description: a string, a sentence describing what the task is testing for
points_possible: an int, the possible amount of points that could be earned for the task
points_earned: an int, the amount of points a student earned on the task
reason_for_score: a string, the reason a student earned a particular grade on the task
B. CS2303_autograder.py

```python
# File Name: CS2303_autograder.py
# Author: Jonathan Morse

# Used to access command line arguments
import sys
sys.path.insert(0, './')
from autograder_core import *
# Allows shell commands
import subprocess
# Allows access to file system
import os

# def UI():
#     print("\nCS2303 AUTO GRADER\n")
#     TA_first_name = input("What is your first name? ")
#     TA_last_name = input("What is your last name? ")
#     AG_of_choice = input("What programming assignment is this for? ")

compile_doc = open("./Output_FILES/did_not_compile.txt", "w")
clean_doc = open("./Output_FILES/not_clean.txt", "w")

# Unzips all student files
unzip_file = unzip_organize()

# Name of zip file containing all students code
zip_file_name = unzip_file[1]

# Creates a list of all student's names
list_of_students = unzip_file[0]

# Number of students being graded
num_of_students = len(list_of_students)

if AG_of_choice == "1":
    # Deletes the not_clean.txt file because it is not needed for PA1
    clean_doc.close()
    subprocess.run("rm ./Output_FILES/not_clean.txt", shell=True)

    for x in range(0, num_of_students):
        # Moves student files into autograder folder
        subprocess.run("mv ./Insert_Zip_File_Here/%s/%s ./PA1/\ 
                        % (zip_file_name, list_of_students[x]),
                         shell=True)
        # Runs autograder for each student project
        subprocess.run("cd ./PA1/; python3 PA1_autograder.py %s %s %s;"%
                       % (TA_first_name, TA_last_name, list_of_students[x]),
                       shell=True)
```
C. User Guide

We developed the CS2303_AutoGrader in an Ubuntu 16.02 virtual machine, and we recommend you use CS2303_Autograder with Ubuntu 16+. We have not tried using Windows or MacOS. We also used the “root” user account within Ubuntu when using CS2303_AutoGrader, so we cannot guarantee the system works using a standard user account.

To download application:
Go to [https://web.wpi.edu/E-project-db/E-project-search/search](https://web.wpi.edu/E-project-db/E-project-search/search) and toggle to “Author” under the “Fields” column and type “Jonathan Morse” under the “Fields” column in the same row. Then click on the project titled “CS2303_AutoGrader” and download the zip file called “CS2303_Autograder.zip”.

**To use:**

Step 1: Unzip and open “CS2303_Autograder.zip”

Step 2: Type “cd src; cd Insert_Zip_Files_Here”

Step 3: Insert zip file of student’s projects into this directory

Step 4: Type “cd ..; python3 CS2303_autograder.py”

Step 5: Fill in first name, last name and what programming assignment needs to be graded

Step 6: Once autograder finishes, type “cd Output_Files; cd Rubrics” to see student’s text file rubrics.
D. Programming Assignment Output Specifications

As we mentioned previously, we designed the functions in this autograder to manipulate the output of student’s programming assignments. In order to do that effectively the output of the programming assignments have to be standardized.

For Programming Assignment One:

Example Output:

```
January 2018
Sun Mon Tue Wed Thu Fri Sat
 7  8  9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30 31
February 2018
Sun Mon Tue Wed Thu Fri Sat
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28
March 2018
Sun Mon Tue Wed Thu Fri Sat
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31
```

Notes:

- Exactly one blank line is needed before and after the “<month> <year>” line, as seen in the example above
- Every line that has the month on it must also have the year on it (ex. “March 2018”)

48
- No blank lines between “Sun Mon Tue Wed Thu Fri Sat” and “ 1 2 3 4 5 6” lines

For Programming Assignment Two:

Example Output:

```
000
0X0
000
000
000
000
000
```

The Game of Life ran for 1 generations.
The Game ended because all of the cells are dead.

Notes:

- The first array must start on the first line of the output, as seen above.
- There should be exactly one blank line in between arrays
- The last line of the output must describe why the game ended. It must contain one of four possible phrases. Those phrases are “predefined”, “dead”, “steady state”, and “oscillating”.
For Programming Assignment Three:

Example Output:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>65</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>ability</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>able</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>about</td>
</tr>
</tbody>
</table>

-------------

<table>
<thead>
<tr>
<th>6</th>
<th>4</th>
<th>Distinct words</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>79</td>
<td>Total words counted</td>
</tr>
</tbody>
</table>

Notes:

- First individual letter output must start on the first line of the file
- There must be at least three dashes ("---") separating the individual word counts and the total word counts
- There must be a tab in-between the number of occurrences and the words or sentences
- The field width of the number of occurrences of each word or total words should be at least six decimal digits