Tool Demo: A Gamified Interactive Feedback System for Grammar Development

Anonymous Author(s)

Abstract

We describe gtutr, an interactive feedback system designed to assist students in developing context-free grammars and corresponding ANTLR parsers. It intelligently controls students' access to a large test suite for the target language. After each submission, gtutr analyzes any failing tests and uses the Needleman-Wunsch sequence alignment algorithm over the tests' rule traces to identify and eliminate similar failing tests. This reduces the redundancy in the feedback that is given to the student and prevents them from being overloaded. gtutr uses gamification to encourage independent problem solving by students: it gives as little information as possible, and students need to prompt the system for further details such as failing tests similar to or different from already seen tests, or even for hints about rules that are the most likely to contain faults. The system visualizes test outcomes over multiple submissions, helping students to keep track of the effects of their changes as their grammar development progresses. gtutr also tracks the students' information requests and uses this to attenuate marks following an instructor-set penalty schema.

1 Introduction

Grammar development is a core part of many compiler courses taught at universities: students are given an informal description of the target language, for which they need to formulate an appropriate context-free grammar, and then implement a corresponding parser, often using a tool such as ANTLR. Students are often also given some examples to complement and clarify the language description, but thoroughly testing the correctness of the developed grammars and parsers requires large and varied test suites that can, for example, be automatically generated from the instructor's hidden grammar [7, 9]. However, given the size and scope of such test suites, the potential number of failing tests can be vast and overwhelm the students, and the successful development of a grammar may rely predominantly on their ability to sift through the error messages produced by the failing tests to trace the potential faults back to their grammars' rules. This can be a very arduous and frustrating process.

We have developed the gtutr tool to make this process easier. gtutr allows students to submit grammars developed for the ANTLR parser generator [1] and then runs a predetermined set of test cases against that grammar. The tool filters the feedback given to the students and gives additional hints that guide students towards the locations in the grammar that most likely contain the faults. gtutr uses the Needleman-Wunsch sequence alignment algorithm to classify the test cases based on their rules traces, in order to identify and eliminate tests that are caused by similar faults in the grammar. This removes redundancy and reduces the amount of feedback needed to convey the same information to the students. A spectrum-based fault localization [8] approach is then used to pinpoint the locations in the grammar that most likely contain these faults. The tool combines these features into an interactive system that provides an ideally more enjoyable way of developing and debugging grammars.

2 Preliminaries

Basic notation We follow the notation from [4, Section 4.2.2]. A *grammar* is a four-tuple $G = (N, T, P, S)$ with non-terminals N, terminals T, $N \cap T = \emptyset$, productions $P \subset N \times (N \cup T)^*$, and start symbol $S \in N$. We use $\alpha A \beta \Rightarrow \alpha\gamma\beta$ to denote that $\alpha A \beta$ *produces* $\alpha\gamma\beta$ by application of the rule $A \to \gamma \in P$ and use $\Rightarrow^*$ for the reflexive-transitive closure.

The *yield* of a sentential form α is the set of all words that can be derived from it, i.e., $\text{yield}(\alpha) = \{w \in T^* \mid \alpha \Rightarrow^* w\}$. The *language* $L(G)$ *generated by a grammar* G is the yield of its start symbol, i.e., $L(G) = \{w \in T^* \mid S \Rightarrow^* w\}$.

Test suites. A *test suite* consists of a list of test inputs for the unit under test (UUT) and corresponding expected outputs (which can also be a specific system error, e.g., for illegal inputs). The UUT *passes* a test if it produces the expected output for the given input and *fails* otherwise.

In our case, the UUT is a parser, a test input is a word x and the expected output is either *accept* (if the test is *positive*, i.e., $x \in L(G)$) or *reject* (if the test is *negative*, i.e., $x \notin L(G)$). Note that the verdict is *pass* if the UUT rejects a negative test case, i.e., the parser identifies an expected syntax error.

Grammar-based test suite construction. gtutr works over a test suite for the target language that must be provided by the instructor as part of the set-up procedure. The test suite needs to have adequate coverage of the grammar and should ideally contain both positive and negative test cases to test the student grammars thoroughly, and to allow the fault localisation algorithm to operate optimally. The test suite must also be big enough and contain enough redundancy, so that the classification algorithm can return meaningful

SLE'20, October, 2020, Cyberspace
2020. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

results. Test suites that fit these criteria can be constructed manually by the instructor, or can be auto-generated. We use positive test suites satisfy several grammar coverage criteria [7], using a generic cover algorithm [6], and negative test suites are constructed using algorithms proposed in [9].

Spectrum-based fault localization for grammars. Raselimo and Fischer [8] describe a method aimed at identifying faulty rules in a context-free grammar. It takes as input a test suite and a parser's implementation of a grammar under test, instruments the parser to collect grammar spectra (which are sets of rules applied during paring) and gives an output of a ranked list of suspicious rules.

Grammar spectra for individual tests are correlated with each test outcome and aggregated into four basic counts for each grammar rule r: $ep(r)$ resp. $ef(r)$ are the number of passed resp. failed tests in which r is executed, while $np(r)$ resp. $nf(r)$ are the number of passed resp. failed tests in which r is not executed. A suspiciousness score is then computed and assigned to each rule r via defining some ranking metrics using these basic counts. Higher scores indicate higher bug likelihood. This method has been adapted with minimal changes from software fault localization techniques, i.e., by replacing the notion of 'executed' program statements with that of 'applied' grammar rules. Wong et al. [10] gives a good overview of software fault localization techniques.

3 Test case classification with the Needleman-Wunsch Algorithm

In a large test suite, there may be many test cases that are similar to each other. Merely increasing the number of failing test cases shown to a student will therefore not necessarily give them additional information to help them find a fault in their grammar, e.g., if enough similar test cases have already been shown. On the other hand, presenting only *different* failing tests is not necessarily the right approach either, because students may need different similar tests to form and confirm (or disprove) a hypothesis about the fault location.

We therefore need to find out which test cases are similar, and indeed to quantify this similarity, so that students can decide how many similar test cases to be shown, and so to maximise the value of the feedback. To this end, gtutr classifies test cases using a modified version of a global sequence alignment algorithm called the *Needleman-Wunsch* [5] algorithm.

We define a test's *rule trace* as the sequence of rules that are applied when a test is parsed. When a test is accepted by the parser, its rule trace consists of the entire sequence of rules in the order in which they were applied (which for an LL-parser means the order in which the parse functions were called). When a test is not accepted by the parser, the test's rule trace consists of all the rules that have been applied to the left of the error position. Whether a test case is positive or negative is irrelevant in this computation, as each trace is only dependent on whether a test case is accepted or rejected by the parser. Classification on these traces is done over the entire test suite, and decisions on which test cases must be displayed to the student are made later.

```
program  body     vardecl  varseq  type
program  funcdef  -        varseq  type
```

Figure 1. Possible alignment of two rule sequences by the *Needleman-Wunsch* algorithm for a toy language.

We can see in Figure 1 two possible rule traces for two different test cases for a toy grammar as seen in Figure 4. Both tests entered first at rule `program` and entered last at rule `type`, at which point an error was produced. However, their rule traces are not identical. To quantify the similarity between different rule traces, the Needleman-Wunsch algorithm is then used on these sequences. This is a dynamic programming algorithm that quantifies the goodness of the alignment over the entire length of two sequences. It consists of three steps:

1. Initializing a score matrix.
2. Calculating scores and filling in a trace-back matrix.
3. Calculating the alignment from the trace-back matrix.

Given two sequences, a score must be associated with each possible alignment between them. The score of an alignment consists of a sum of scores for each corresponding rule in the aligned sequence. Matching rules increment the score, mismatching rules decrement the score, and gaps in the alignment (i.e., locations in the alignment where there is no rule for one of the sequences) penalize the score heavily. For example, in Figure 1, the first rules will match, the second will be a mismatch, and the third will be a gap.

Given this scoring scheme, the score matrix C can be initialized. For two sequences $a = a_1a_2...a_n$ and $b = b_1b_2...b_3$, the score matrix C is a $i \times j$ matrix where any cell $C(i, j)$ is the maximum value of

$$C(i,j) = max \begin{cases} C(i-1, j-1) + S(a_i, b_j) \\ C(i-1, j) + g \\ C(i, j-1) + g \end{cases}$$

where $C(i, j)$ is the score at position i in sequence a and position j in the sequence b. $S(a_i, b_j)$ is the score for aligning the characters at positions i and j. g is the gap penalty. In other words, the value of a cell $C(i, j)$ is only dependant on the values of the adjacent northwest diagonal, up and left cells. As the score matrix is created, the trace-back matrix is filled up with `diag`, `up` or `left` values. After the matrices have been filled up, the trace-back matrix can be used to deduce the optimal alignment by starting at the bottom rightmost element and traversing according to the values in the array. The value `diag` in position (i, j) in the trace-back matrix means that the letters a_i and b_i are aligned. `left` means that a gap was introduced in the left sequence, and `right` means that a

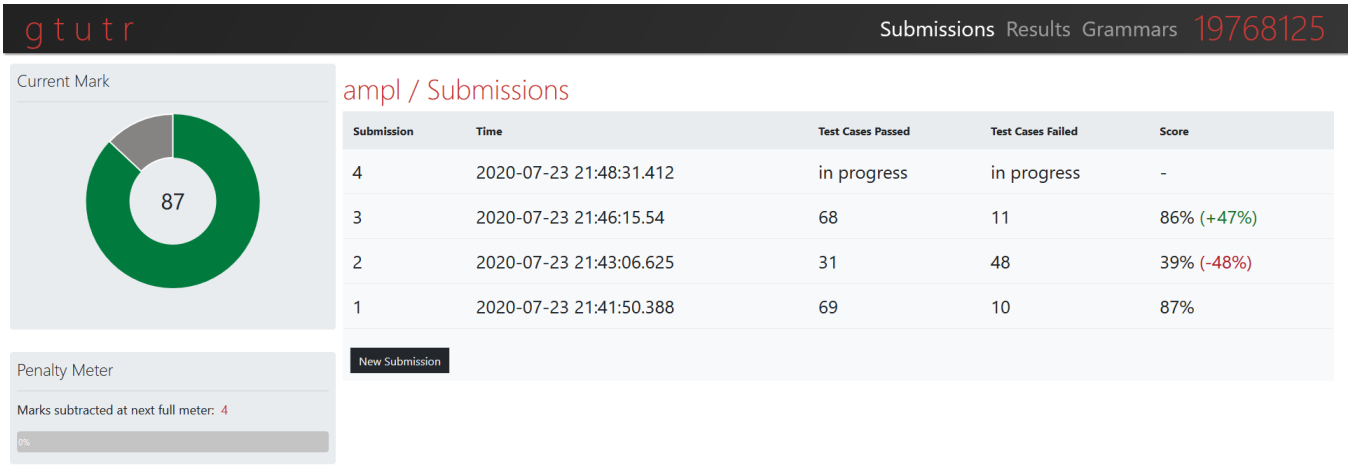

Figure 2. Student dashboard showing submission history.

gap was introduced in the right sequence. The adjusted sequences are built up from the values in the trace-back matrix, and the final alignment score is calculated.

Tests that induce rule sequences with high alignment scores are classified together, while those that do not have sufficiently high alignment scores with any other sequences form new error classes. In this way, test cases of a grammar can be classified according to their rule traces.

Note that in gtutr, the alignment algorithm classifies the test cases according to their outcomes in the student's latest submission. Therefore classifications will change over time as the student makes changes to their submissions.

4 Tool Design

Overall Approach. gtutr was designed as an interactive educational tool for use by both students and instructors. The main design goal was to guide students towards improving their grammars; it therefore gives out less information than the student might want, to encourage and foster independent problem-solving.

The classification described in the previous section assists the tool in deciding on which test cases to display, but the tool also applies other metrics to make the feedback as diverse as possible. It attempts to include both positive and negative test cases if possible, as well as failing and passing test cases from the same class. Another feature is a penalty system that aims to discourage students from "teasing too much information out of the system". For each action taken by a student that requests the disclosure of additional information, the system assigns a penalty value. These values represent percentages of a penalty meter that, when filled to 100%, subtracts a predetermined amount of marks from the student's score. There are three main views for student interaction, which we will describe below.

Main Student Dashboard. Students can upload their grammars to gtutr, which will fetch and run untested grammars using the provided test suite. When a student logs in, they can select a grammar project and are then redirected to the 'Submissions' page. This view is shown in Figure 2.

The circular graph on the left displays the current mark for a student. It will also display the total penalties in red if the student has any. Below this is the penalty meter, which displays the current running total of penalties as well as the amount that will be subtracted from the total score once the meter is filled.

The table in the middle of the page displays a student's submissions, both past and in progress. The total number of failed and passed test cases are shown, as well as the time of submission and the improvement from the previous submission. Clicking the 'New Submission' button will bring up a window where a grammar file can be selected and uploaded. The top right corner contains links to the other pages that the student has access to.

Detailed Feedback View. Clicking on 'Results' will redirect the student to the detailed feedback made available by the system, as seen in Figure 3.

The feedback is presented as a small table, where each row displays the test case's name (which could be mnemonic to give more hints) and the test's status in each of the student's submissions. Clicking on the test's name will display its contents, and clicking on the green (passed) or red (failed) blocks will reveal the error message associated with the test case for that specific submission. This can also be seen in Figure 3: the last two rows of the table represent the content and error message respectively for the test case in row 6.

Initially, students are only presented with both a small number of error classes and a small number of test cases per class. This is designed to encourage the student to attempt to solve the problem with limited feedback. If a student wants

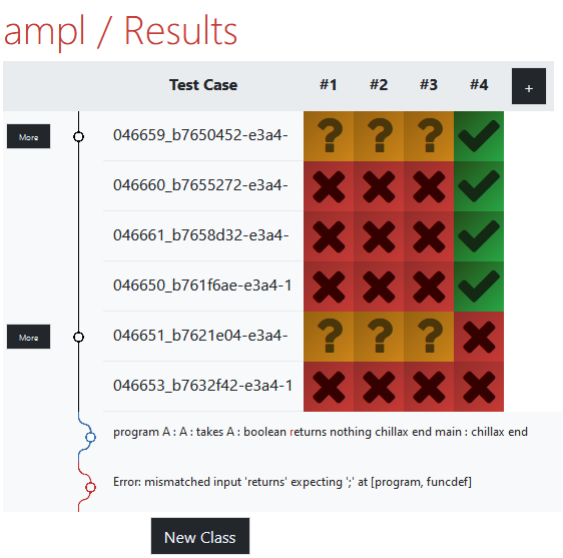

Figure 3. Student result view. Each row in the table represents a test case and its performance over a student's submissions, which span the columns. Green blocks are passing test cases, red blocks are failing test cases, and orange blocks denote an 'unknown' status that appears when a student has requested to view a previously unseen test case. For such a test case, clicking on its unknown blocks can reveal the actual status. Clicking on the 'More' and 'New Class' buttons will load a new test case from an already displayed class or an unseen class, respectively.

more feedback, they can elect to be shown more test cases in a specific class by clicking the 'More' button, or an additional previously unseen class altogether with the 'New Class' button. If a student has submitted multiple grammars to the system and requests to see a previously unseen test case, the status of the test case will only be known for the latest submission, and the status for each other submission is unknown, as represented by the orange blocks with question marks in Figure 3. Students can elect to uncover the status of a test case from a previous submission if they wish by clicking on the block.

All of these additional features are explained with tooltips and each action that incurs a penalty will have a tooltip disclosing the exact amount that will be added to the penalty meter, so that students are aware of the cost of each action.

Localization View. When a student clicks on the 'Grammars' link, they will be redirected to a page as can be seen in Figure 4. This is where students can view the most suspicious rules of their grammars. When pressing the 'Find faulty rules' button for the first time, the tool will give them the top most suspicious rules in no particular order. Only the Top-5 rules will be shown, as long as they have a non-zero suspiciousness score, and are therefore seen as suspicious by the localization algorithm. This can be seen on the left-hand side of Figure 4, where only two rules have been flagged as suspicious.

With each successive request to view the faulty rules, the tool will start to reveal the order of their suspiciousness, starting with the least suspicious rule, as seen on the right-hand side of Figure 4. The darker the red color, the higher ranked a rule is. The rank for a rule can also be seen by hovering over its production. As an example, in Figure 4, the faulty rule in the submission on the left is `funcdef`. In the right-hand submission, the faulty rules are `varseq` and `output`.

Instructor View. The instructor is able to create new grammar projects, which includes supplying the test suite as well as calibrating the penalty values associated with each penalizable action. The test cases are configured by the instructor to be positive or negative, and they can be hidden from the students' result section if the instructor does not want to make all the test cases available to be seen. The instructor also has the ability to view statistics on the students, their submissions as well as the test cases. A summary of the flow of the program can be found in Figures 5 and 6 in the Appendix.

5 Tool Implementation

gtutr is implemented in Java as a Java Spring Boot [3] application and uses the Spring MVC framework for the web interface. The database storing the submissions, grammar information and results are hosted on Amazon Web Services (AWS). The server runs two components simultaneously, that is the calculation and extraction of the feedback and its display in the web application, and the execution of the ANTLR components in the background.

ANTLR parser execution. gtutr uses the ANTLR v4 library [1] for Java, which includes the `antlr4-runtime` and `antlr4` Maven dependencies. The Java library allows manual invocation of a parser as well as the automatic generation of a tree walker that can be used to visit the nodes of the tree that was generated by the parser. The application uses the tree walkers to extract the rule traces of the test cases for classification and fault localization purposes.

The ANTLR-parser invocation requires the presence of the parser, lexer and other related classes in the tool directory. These files are usually automatically built by Maven in the compilation stage of its life-cycle, but this requires an ANTLR grammar file to be present in the project. Since gtutr fetches the grammar files from the database and compiles them continuously, this is not ideal. A dynamic class loader and compiler was therefore integrated with the ANTLR code to generate the needed files on the fly using only the student's grammar submission fetched from the database.

Figure 4. Part of the page seen by the student when they want to view their grammars and the possible faulty rules. Suspicious rules have been highlighted for each grammar. Initially each suspicious rule is shown as equally suspicious. Clicking on 'Find faulty rules' will start to reveal the ranks of the rules, with the redness of the rule indicating its suspiciousness.

The part of the application responsible for the generation of a student's parser and its subsequent testing is contained within a thread that monitors the database for any new submissions. The tool fetches the new submission from the database, generates a parser, and fetches the test cases from the database, running through them each individually, while keeping track of failures, error messages, and rule stacks. All this information is then again saved to the database where it can be used to display feedback to the student and instructor.

Web interface. The web application makes use of Java Server Pages (JSP). JSP is a collection of technologies that assists with the creation of dynamic web pages implemented (in this case) in HTML. JSP is similar to PHP but uses Java as a programming language, which makes integration with the main application easier. The application also makes use of Asynchronous JavaScript and XML (AJAX) to dynamically update the web page as needed.

CAS. To restrict access to the web application to members of staff and students only, we integrated a central authentication service (CAS) using Spring Security, which allowed us to use our University's single sign-on service to manage access to the application and the data.

JPA. The application uses the Java Persistence API (JPA) [2], which is a Java specification for accessing, persisting and managing data between Java object/classes and a relational database. JPA is used as it provides database abstraction and the automatic handling of data types. Queries do not need to be written by the user, and JPA provides performance optimizations that speed up typically performance reducing database queries.

6 Conclusion

gtutr is an interactive feedback system that was designed to assist students in developing context-free grammars and their corresponding parsers. It analyzes failing tests and uses the Needleman-Wunsch sequence alignment algorithm over the tests' rule traces to identify and eliminate similar failing tests. The tool then provides highly interactive and visually stimulating feedback to the student in the form of test outcomes and grammar fault localisation as well as a point and penalty system.

gtutr has been designed to use ANTLR and its components, but can be adapted to work with other parser-generator systems by adding a simple localised component in most cases. The notion of rule traces and their computations would also need to be adapted for such modifications.

Currently gtutr is in a trial stage and is being tested on a small group of students for an SLE course at our University.

References

[1] [n.d.]. ANTLR v4 Source Code. https://github.com/antlr/antlr4.

[2] [n.d.]. JPA Source Code. https://github.com/eclipse/javax.persistence.

[3] [n.d.]. Spring Source Code. https://github.com/spring-projects/spring-framework.

[4] Alfred V. Aho, Monica S. Lam Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (Second Edition).* Addison-Wesley.

[5] Zhihua Du and Feng Lin. 2004. Improvement of the Needleman-Wunsch Algorithm. In *Rough Sets and Current Trends in Computing*, Shusaku Tsumoto, Roman Słowiński, Jan Komorowski, and Jerzy W. Grzymała-Busse (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 792–797.

[6] Bernd Fischer, Ralf Lämmel, and Vadim Zaytsev. 2011. Comparison of Context-Free Grammars Based on Parsing Generated Test Data. In *Software Language Engineering - 4th International Conference, SLE 2011, Braga, Portugal, July 3-4, 2011, Revised Selected Papers (Lecture Notes in Computer Science)*, Anthony M. Sloane and Uwe Aßmann (Eds.), Vol. 6940. Springer, 324–343. https://doi.org/10.1007/978-3-642-28830-2_18

[7] Ralf Lämmel. 2001. Grammar Testing. In *Fundamental Approaches to Software Engineering, 4th International Conference, FASE 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings (Lecture Notes in Computer Science)*, Heinrich Hußmann (Ed.), Vol. 2029. Springer, 201–216. https://doi.org/10.1007/3-540-45314-8_15

[8] Moeketsi Raselimo and Bernd Fischer. 2019. Spectrum-based fault localization for context-free grammars. In *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2019, Athens, Greece, October 20-22, 2019*, Oscar Nierstrasz, Jeff Gray, and Bruno C. d. S. Oliveira (Eds.). ACM, 15–28. https://doi.org/10.1145/3357766.3359538

[9] Moeketsi Raselimo, Jan Taljaard, and Bernd Fischer. 2019. Breaking parsers: mutation-based generation of programs with guaranteed syntax errors. In *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2019, Athens, Greece, October 20-22, 2019*, Oscar Nierstrasz, Jeff Gray, and Bruno C. d. S. Oliveira (Eds.). ACM, 83–87. https://doi.org/10.1145/3357766.3359542

[10] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Trans. Software Eng.* 42, 8 (2016), 707–740. https://doi.org/10.1109/TSE.2016.2521368

Appendix

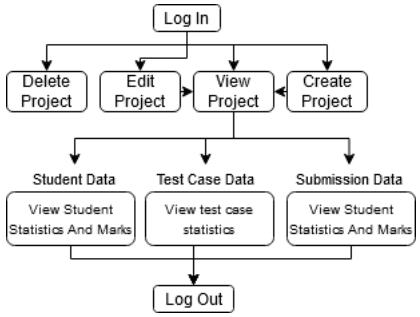

Figure 5. Flow diagram of instructor actor

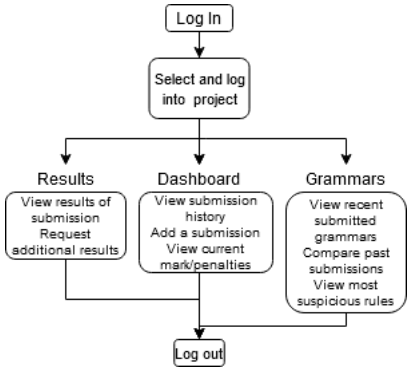

Figure 6. Flow diagram of student actor