A proposal for Automatic Evaluation of a Compiler Construction Course

Emilio Julio Lorenzo Galgo
LTCS group at UNED
C/ Juan del Rosal, 16
28040 Madrid Spain
+34 91 398 8652
emiliojulio@lsi.uned.es

Javier Vélez Reyes
LTCS group at UNED
C/ Juan del Rosal, 16
28040 Madrid Spain
+34 91 398 4976
jvelez@lsi.uned.es

ABSTRACT
Experience in teaching the subject of Language Processors within the degree for Technical Engineering Computer Science, highlights the difficulties developing a compiler, for both students and teachers. The former because they are often disoriented when they are unable to get some feedback on their work and the latter because of the amount of effort and time involved to perform the evaluation. This paper presents a system for helping evaluation using test cases, aligned with the trends of EHEA (European Higher Education Area) ongoing evaluation methodology. This approach is implemented in three phases where different problems are faced. The first phase consists of designing a test case set, accurate enough to discern whether the student successfully absorbed the different concepts and contents of the subject. The second phase involves the implementation of a runtime environment for the designed tests, so that many of the tasks required by the teaching staff can be automated. In the third phase automatic assessment according to different criteria is implemented. The fourth phase is reporting the results of the previous execution like straightforward feedback for both students and teachers.

Categories and Subject Descriptors
K.3.2 [Computers and Education]: Computer and Information ScienceEducation – Computer science education, Curriculum
K.3.1 [Computer Uses in Education]: Computer-assisted instruction(CAI), Distance learning
D.3.4 [Programming Languages]: Processors – Compiler, Code generation

General Terms
Management, Measurement, Languages, Human Factors

Keywords
Compiler automatic assessment feedback

1. INTRODUCTION
Adaptation to EHEA\(^1\) methodologies requires the inclusion of practical exercises, as well as ongoing evaluation throughout the academic year. This trend includes project-based learning (PjBL)[10], which is based in the use of in-depth projects to facilitate learning and assess student competence. This approach fits with Language Processors course, eminently practical, where students have to design and implement a full compiler for a programming language proposed by the teachers. The main problem here is: How do teachers manage to evaluate using ongoing methodology? In situations with hundreds of students, teachers have several problems related to time and effort for the assessment and feedback of the student improvements. These problems are compounded in the case of distance learning, where teacher-student communication is usually performed only by email.

There are several limitations to traditional evaluation systems in case of compilers like in [1]. Firstly, the kind of work, which students have to perform, changes every year, adding and removing functionalities and concepts, and not only a mere change in the instructions. Secondly, we have to evaluate the compiler itself. However, the traditional systems are focused in the final code, but not in the processes to create it. Finally, the compiler has to check source code errors, what means “erroneous programs” should be marked as correct, if the error is detected.

There is a lot of work related to building compilers like [2] [7], but the aim of this work is to help in the evaluation independently of the compiler tools used. For this evaluation, the chosen method is based on test cases, comparing the final and partial outcomes of the compiler in order to grade students and give feedback. Several approaches propose automatic generation of test cases [3] [4], but they do not usually deal with automatic grading and feedback. Other problems are the complexity of the systems, which make them difficult to use, and the dependency of certain architecture.

In this paper, we propose an automatic evaluation system that: 1. indicates the state of the compiler, 2. helps to grade it and 3. provides convenient feedback about their mistakes and successes.

Our proposal involves several recommendations to design test cases and an easy-to-use automatic evaluation system, independently of the compiler tools used. We use “evaluation" instead of “grading” because our goals are not only to assess but also to give feedback to students and teachers. This system has been implemented in order to evaluate the stage of development

\(^1\) http://www.ehea.info/
of a full compiler, which includes the usual phases: lexical analyzer, syntax analyzer, semantic analyzer and intermediate and target-machine (final) code generation. Each part will be evaluated separately, so one of the prerequisites is that they have to be implemented in separated modules.

In the following sections we discuss how teachers must design the test cases, how the system runtime environment was implemented, what kind of automatic assessment we propose, the report generation and technical issues of our system. To finalize we will present the system used in a real scenario and conclusions.

2. THE SYSTEM ARCHITECTURE

Figure 1 shows the system architecture. Our system manages the flow-control of the compiler architecture, giving test cases as inputs, collecting outputs and giving final code to the emulator.

![Figure 1. System Architecture](image)

We use black-box evaluation for checking the outcomes of each phase. Therefore, the different modules must be able to inform the system about several events, basically writing messages in log files, which will be compared with the expected results. Finally, the execution of final code has to generate a text format outcome as a result of this execution. In the rest of the paper we refer this final code to be assembly language, but it is not always necessary.

3. DESIGNING TEST CASES

Teachers must design test cases for the evaluation of student knowledge and for checking whether they have assimilated the theoretical concepts of the subject by putting them into practice building the compiler. This task has to be redefined annually by adapting tests to the proposed programming language. Along with the implementation of the source code files, the expected results must be also written. For instance, one year the proposed language accepts only subtraction. Examples of errors are the following:

<table>
<thead>
<tr>
<th>Error Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic Arithmetic</td>
<td>Type or variable value assignation, recursive subprograms, flow-of-control statements, subprograms (functions, procedures and methods) declaration, precedence of operators, type expressions and data structures, erroneous nested comments, bad symbols.</td>
</tr>
<tr>
<td>Advance Arithmetic</td>
<td>Logical variables declaration, logical operators, logical variables declaration, logical operators, recursive subprograms, erroneous nested comments, bad symbols.</td>
</tr>
<tr>
<td>Semantic errors</td>
<td>Type or variable value assignation, procedures when expecting functions, return in functions and special language characteristics.</td>
</tr>
<tr>
<td>Non-erroneous programs</td>
<td>Erroneous programs, such as: Type or variable value assignation, procedures when expecting functions, return in functions and special language characteristics.</td>
</tr>
</tbody>
</table>

A useful method for designing test cases is that they must be as atomic as possible. That is, we must avoid, as far as possible, to include additional functionalities in a specific case because it could be misleading when the test fails. For example, in most test cases it is necessary to add a variable declaration and, if that functionality is incorrect, the test will fail, but does not mean that the concept to evaluate does not work. For this reason, test cases are designed with increasing difficulty. If the first set of test cases fails, there is no point in further evaluation. Another consideration to take into account is the fact that we have to check the different phases of a compiler, which in our case is divided into lexical, syntactic, semantic, code generation, intermediate and final code.

3.2 Erroneous Programs

An important set of tests is related to detect coding errors in each stage or phase. This means that the compiler works properly when detecting the errors in “erroneous programs”. To face the problem of how to identify the error and in which phase it happens, we define a standard error messages that students have to emit as output of their compilers. In [8] is showed that, although there are a large number of different error types, several categories are prevalent. In our system, the main errors are the following:

- **Lexical errors**: Non-closed comments, erroneous nested comments, bad symbols.
- **Syntax errors**: Incoherent patterns, types or variables declaration, block construction.
- **Semantic errors**: Type or variable value assignation, procedures when expecting functions, return in functions and special language characteristics.

This information along with the line and column number is also useful in case of a non-erroneous test case, because it will show what kind of wrong error is detected by the compiler and in which phase of development is located. This is an important feature in order to provide debugging information to students. Experience shows that in many cases it is difficult to determine the error, looking at the output of the tools used to develop the compiler. For example, the “+” symbol is not allowed in the case our language accepts only subtraction. Examples of errors are the following:

```
[SYNTAX ERROR]-Token[34] [‘+’, 10, 13] - Block Error
[SEMANTIC ERROR]-Token[56] [‘b’, 20, 14] - Type Error
```
4. RUNTIME ENVIRONMENT

The evaluation system automates all necessary steps to generate the compiler and runs it with the previously designed test case sets as inputs. The first step is to prepare the environment where compilers will be evaluated. To do this, each student has their own directory, including special directories to contain test cases and results. In this sense, the test case set is collected from a specific general directory and then copied to each student’s. This solution allows teachers to modify test case set in two easy ways: all students, changing the general test case, or a single one, changing the student directory. This differentiation is important because, sometimes, it is easier to adapt the test cases to a special compiler instead of changing the compiler itself. In so manner, if the compiler has a little mistake, we can go on just changing the test to avoid it, thus giving more feedback about other functionalities. Once the environment is ready, the steps that the automatic evaluation system carries out are the following:

1. Compiling the compiler. The compiler is also a program that must be compiled and executed. This step is only performed once for each student and if it fails, the evaluation stops here.

2. Compiling the test case. The compiler compiles the test cases. The different phases of a compiler will be checked in this step. If successful, the result will be a final assembly code file.

3. Running the final code. Usually using an emulator to execute the final assembly code created in the previous step. If successful, the outcome will be a text file with the result of the execution.

4. Evaluating the outcome. In this step the previous text file is compared with the expected result. For example, if the source code is to make the sum $2 + 2$, the result should be 4. We have to assure the compiler works and that it works correctly.

5. Reporting generation and grading. Steps 2 to 4 have to be repeated for each case of the test case set. Once all cases have been checked, relevant reports are generated and a provisional student assessment is also calculated.

To illustrate all this steps, Figure 2 shows an example where test case number 2 fails in the compilation phase. This case is related to i/o functions and watching the compilation report we find out that the error is an exception in the method “getSymbolFactory”, which is related to make a new symbol table and it causes a false lexical error. Those reports will be explained in report generation section.

5. AUTOMATIC ASSESSMENT

Multiple proposals provide automatic assessment [1] [9], but the goals of a Language Processors course are different from a programing course, so we do not want to check the code for being well formatted, we are interested in the results of this code. For instance, the Symbol Table must do what we expect, but it implementation is not too important. Besides, the complexity of the compiler and the variety of implementation techniques make us discard a deeper code style evaluation.

A simple approach for the grading method follows the classical formula:

\[ \text{mark} = \frac{\text{Success test cases}}{\text{Total test cases}} \]

In this approach, a test case is considered success if the execution succeeds and returns the expected value. This is carried out by comparing the outputs generated with the expected. [6] points out that a success is determined in two dimensions: correct behaviour and user-friendliness, where the latter is largely determined by the helpfulness of error messages. We can check this dimension due to the standard vocabulary for errors described previously.

Although the emulator outcome (final result) format is described by the assembly emulator, there are many instances in which students are unable to follow, either by neglect or inability. For example, the output should be lowercase, but many practices use capitals. We use a text filter, prior to the result matching, to avoid generating false errors in those cases.

Our proposal allows a more sophisticated method, using the features when compiling and running the student compilers for a more accurate assessment. Specifically, the assessment is divided into three distinct parts:

- **Compilation.** The student compiler compiles the source code of a test case correctly. This means that there are no errors on the lexical, syntax and semantic analyzers. This part corresponds with step 2 of section 4.
- **Execution.** The compiler generates intermediate code and final code in the form of text files containing the program in assembly language. This part corresponds with step 3 of section 4.
- **Correction.** The assembly code runs on an emulator and produces the expected result. This part corresponds with step 4 of section 4.

With these changes the grading method may be more complex. An example of the new calculation is as follows:

\[ \text{mark} = \frac{\sum_{i=1}^{N} C_i \cdot H_i}{N} \]

Being $N$ the total number of test cases, $C_i$ the numerical result of evaluating a test case and $H_i$ the weight of the case. $C_i$ can take four possible values: 0 if it does not compile 0.25 if it only
compiles, 0.4 if it compiles and executes, and 1 if the outcome is correct. \( H_t \) depends on the test importance, therefore important or advanced concepts can improve the final marks more than initial building compilers concepts.

This formula could be improved to be more specific, for instance, by adding lexical, syntax and semantic phases, or by considering the kind of errors. Another easy way to implement weights, increasing the depth of the evaluation, is to add more test cases related with relevant functionalities.

In this point we also have to refer to “errorneous programs”. If one of those tests fails, it is considered right compilation, execution and correction, without checking the last two assessment parts. For helping teachers, it is possible to say to the system what kind of error we are expecting, in a declarative way using the standard error messages described previously.

6. REPORT GENERATION

An important part of the project is the automatic report generation for both students and teachers. These reports are made by analyzing the logs generated at each assessment phase (compilation, execution, correction). Visual comprehension prevails for helping to understand them by using Web page format reports. For teachers, a comprehensive report is generated, with the identifier of students and their proposed marks. The rest of the reports are for each student personalized monitoring. In particular, four of them are generated:

- **Compilation of the compiler.** This report shows whether the student compiler can be built correctly or not, showing potential programming errors committed.
- **Compilation log.** This report shows the log generated by the compiler during the compilation phase of each test case. In case of error, there is a text indicating its type and where it is located (lexical, syntax or semantic phase). This is possible through the use of a standard error vocabulary (see Figure 2).
- **Final code log.** This report aggregates the results of running each test case final code with the assembly emulator. These results will be compared with those expected and determine if the case is “correct”.
- **Evaluation summary.** This report shows, in a table format, the results of each part of the assessment (compilation, execution, correction) in each test case (see Figure 2).

The compilation and final code log reports along with the test cases are particularly useful for the students and can be consulted by students for an in-depth checking as well as for finding the concepts they need to further improve in order to carry out their practice.

These reports are used not only for debugging but also to exchange them with other students and teachers. Using a common vocabulary messages and reports, it is easy to ask for helping and hints when developing the compiler. According [7] good outcomes promote collaboration between students. Moreover, the reports help teachers in their communication with students. A complete written report, including errors and clarification, is a hard task and the EHEA ongoing evaluation makes it even more difficult. Our experience shows that sending standardized reports is enough feedback in most cases.

7. TECHNICAL ISSUES

Two prototypes were built incrementally in order to implement the runtime environment. The first prototype was programmed using an operating system scripting language for automation. The run and evaluation phases were implemented by two separate Java programs, called from the script file. This first prototype had several important problems: First, by blocked execution. When the compilation or execution of a student compiler generates an error or unexpected exception and blocks the automation. Second, by little flexibility. It is difficult to change the behaviour of automation, in both execution and evaluation. This may be interesting due to students make unexpected changes in their compilers that can hinder or prevent their evaluation. Finally, system dependency. Because of operating system scripts.

To solve these problems, the second prototype was developed. It replaced the operating system scripting language and JAVA programs by Groovy\(^2\), a JAVA-based scripting language. With this change flexibility was achieved to treat particular cases easily and all steps are unified in a single program to facilitate maintainability. Thanks to the new environment, it is possible to avoid blocking practices. In this way, all compilers can be evaluated at once, adding a new level of automation. Not only a practice is evaluated but also all of them in a batch process. The processes of compiling the compiler and running its test cases is carried out using tasks programmed in ANT\(^3\). We defined a task for each phase to check.

Using those technologies the system is able to be run in a web container, so it can be used as an online evaluation server service. Students can submit their work and get almost instant feedback.

8. TESTING THE PROPOSAL

This automatic evaluation proposal was tested in the context of a Technical Engineering in Computer Science distance degree, in a language processors course with about two hundred students. Two different statements were developed to discourage plagiarism. The system was used for helping grading student compilers and as feedback to students who wanted a review. We used JFlex\(^4\) and CUP\(^5\) as lexical and syntax generators along with our own compiler development architecture. To check the final code we chose an assembly emulator that allows windows and console running. The latter is important for automation, because accepts text inputs and generates text outputs, what it is necessary for matching them with the expected results.

The testing results are interpreted from a qualitative and quantitative point of view. The former shows how students and teachers see the new work environment and the latter presents some data to support these findings.

8.1 Qualitative

The evaluation and further communication with the student is greatly simplified and reinforced by the student emails. They show their positive evaluation of the test case set and associated reports tailored to the idiosyncrasies of the subject. This is also exposed in the forums posts, where students interact with each other.

\(^2\) [http://groovy.codehaus.org/](http://groovy.codehaus.org/)
\(^3\) [http://ant.apache.org/](http://ant.apache.org/)
\(^4\) [http://jflex.de/](http://jflex.de/)
other more fruitfully. They can understand better their classmate’s problems and can dialog with the same vocabulary, what improve collaborative learning. The practice instructions became also more comprehensible. The questions about the functionalities that the proposed language has to implement are usually answered just looking at the test cases. In this manner now, the student questions are focused in the concepts about building compilers more than in “what they have to do”. Concerning teachers, they have the same advantages about their communication with students; furthermore, they can follow easily the student improvements just checking the historical of evaluations.

8.2 Quantitative

It is estimated that in the case of practices with errors, the evaluation and reporting time is reduced almost entirely. In previous years the teacher had to test practices one by one and answer individually to each student.

An indicator that can be used to measure the usefulness of the reports when reviewing is the number of messages sent by students and teachers. In the case of students who were not sent the reports, it shows an average of eight sent messages until the student understands and accepts the evaluation. In the case of those who were sent them, the average is two. It is noted that students accept their marks and need less interaction to understand their mistakes. The consequence is a reduction of debugging time by both the student and the teacher, so communication between them is more direct, clear and concise.

9. CONCLUSIONS AND FUTURE WORK

This paper has presented a proposal for automatic evaluation of the practical part of a Language Processors course within a distant course computer science degree and focused on the trends of EEES ongoing evaluation methodology. The approach used to assess students’ knowledge is the realization of a practice throughout the course, following the project-based learning philosophy. Practice to do is to implement a full compiler for the programming language proposed by the teaching staff each year.

We have seen that this sort of practice is quite complex to implement and therefore, to assess. Our system helps to automate all processes and phases involving compiler evaluation. In order to help teachers to test compiler functionalities, a basic test cases set has been defined. We have also proposed different phases and steps to be carried out by teachers in order to check the student compiler. This framework along with common compiler messages vocabulary allows teachers to reduce time consuming and improve feedback and communication to students. Finally, the proposal was tested in a real distance course, showing that both students and teachers agree with it.

There are several topics that can be tackle in the future. Firstly, we should try other ways to automatic assessment, using other formulas of the state of the art. Another important functionality is to incorporate a plagiarism detection mechanism by comparing the logs obtained. For a complete evaluation, our system must be tested with other compiler tools and architectures and, finally, the system should be integrated in a LMS, so that eLearning standards could be implemented.

10. ACKNOWLEDGMENTS

Our thanks to UNED for the support of III educational innovation network, Ministerio de Ciencia e Innovación for the support of Project TIN-2009-14317-C03-03 (http://crease.lsi.uned.es), Comunidad de Madrid for the support of E-Madrid Network of Excellence S2009 TIC-1650 (http://www.emadridnet.org). Also, our thanks to Sergio Galvez for previous work.

11. REFERENCES


