Generative Learning Object Assessment Items for a Set of Computer Science Disciplines

Ciprian-Bogdan Chirila[1]

Abstract – Learning objects with static content are good for learning and practice but not very recommended for assessment. The main problem is with content repetition which: enables mechanical answer memorization by the student and replication of answers from class neighbors which is considered as an examination fraud. The Generative Learning Object (GLO) is an evolved concept of learning objects (LO) based on reusing the learning patterns. Enhancing GLOs with dynamic content could increase their reusability even more.

Keywords – blended learning, generative learning objects, generative techniques.

1 Introduction

Generative learning objects [5] are learning objects with dynamic content where the learning pattern can be easily reused [4]. Learning objects usually contain: content items, practice items and assessment items. In this paper we will present the main principles in the implementation of experimental GLO assessment modules created for different computer science disciplines like: i) data structures and algorithms; ii) fundamentals of programming languages; iii) compiling techniques; iv) operating systems.

The first discipline contains two modules DSA1 and DSA2 where are handled: i) search algorithms, sorting algorithms, linked lists, hash tables in the former; ii) trees and graphs in the latter, using random data sets. The second discipline contains one module FPL deals with basic functional programming concepts namely list exercises based on generative trees. The third discipline contains one module named CT which basically deals with the generation of grammars with lexical rules and syntactical rules. The fourth discipline has one OS named module which facilitates the learning of Unix commands. All the GLO items presented can be used for both practice and assessment purposes.

The motivations behind our approach are multiple: i) students tend to use more and more gadgets like smartphones, tablets, tablet PCs, laptops, thus becoming "digital students" [3]; ii) the IT industry nowadays is more expending so computer science disciplines are more and more important in this context; iii) the number of students delivered by universities for the IT industry is quite low especially in eastern-European countries, loosing contracts, thus affecting the national economy.

[1] Lecturer at University Politehnica Timişoara, Romania,
e-mail: chirila@cs.upt.ro, tel. 0040256404061

The objective of this paper is to present an experimental set of assessment items with dynamic content based on several generative models and to try to extend it to a higher level of generalization for the learning of computer programming.

The paper is structured as follows. Section 2 presents a set of assessment items for the Data Structures disciplines. In section 3 we will show how basic commands can be assessed in the context of Unix [13] operating system. In section 4 we will analyse several types of generative learning items applied on the Fundamentals of Programming Languages discipline. Section 5 presents a generative pattern for grammars generation. In section 6 we will describe the prototype implementation. In section 7 we will present related works. Section 8 concludes and sets the future work.

2 Assessment Items for Data Structures and Algorithms Discipline

In this section we will present the DSA1 [10] and DSA2 [15] modules containing GLO items. For starters, we decided to address the data structure discipline because we consider that it has a decent level of complexity, higher than computer programming disciplines and also lower than compiler techniques or other more complex computer science disciplines.

We will consider the following general data structures and their accompanying algorithms: i) searching algorithms on arrays; ii) sorting algorithms on arrays; iii) linked lists; iv) general and binary trees; v) graphs, representations and related algorithms.

Firstly, we will present the DSA1 module where the student must write the steps of the some proposed algorithms. For each step the student has to detail: index values, comparisons with their result, array value exchanges and matches.

The first exercise proposes a search in a generated integer array of the middle element. The search algorithm is randomly selected from a list of three algorithms: i) linear search; ii) binary search; iii) interpolation search. For the first search the array can be unsorted but for the second and the third searches they can not. The variability in this item consists in: i) the size of the array; ii) the sorting order of the array with the constraint that for linear search is unsorted while for binary and interpolation is sorted; iii) the position of the searched element.
For this parameter several interesting particular cases from the learning point of view can be considered: i) the middle position when the array size is odd; ii) in the two middle positions when the array size is even and when there is not only one middle position; iii) the first position; iv) the last position; v) the second position; vi) the penultimate position.

The second exercise is about writing the steps of the sorting by insertion algorithm. The variability of this item consists in: i) the size of the array; ii) the order of the array elements (ordered, random, reversely ordered); iii) the direction of the sorting (ascending or descending); iv) the use of linear insertion or binary search insertion as sub-component of the algorithm.

The third exercise deals with sorting by selection and shares the same variability as the second exercise except for the iv) which is replaced by using simple implementation or performance implementation. The two choices are just two algorithm variants which are presented in the face to face lecture by the discipline tutor.

The fourth exercise is about bubble sort and shaker sort which are related sorting algorithms and where we keep the same variability as in the last sorting algorithm.

The fifth exercise is about merge sorting algorithms applied on files. The variability is completed with different variants: i) 3 files merge; ii) 4 files merge; iii) natural merge; iv) the size of the array (8 to 12); v) the integer range of each element (from 0 to 1000).

The sixth exercise is direct substring search. The variability items are: i) the alphabet used for the characters (small or big caps, from A to Z); ii) the size of the string (usually 10 to 16 characters); iii) the size of the model (5 to 8 characters);
The seventh exercise is substring search based on several algorithms: i) Knuth-Morris; ii) Knuth-Morris-Pratt; iii) Boyer-Moore. This exercise is based on random generation of a string and the identification of a substring model inside it. The variability items in this case are the following: i) the alphabet used for the characters (small or big caps, from A to Z); ii) the size of the string (usually 10 to 16 characters); iii) the size of the model (5 to 8 characters); iv) the position of the searched model; v) with or without overflow on the searched string.

Secondly, we will present the DSA2 module dealing with trees and graphs, where we started another set of generic exercises.

The first one deals with the general tree representation based on parent index array. A parent index array is generated randomly and the student has to draw the equivalent diagram tree representation and to write the first-descendant and right-sibling arrays. The generation of the array is based on taking one parent index and to replicate it in the parent array starting at an index higher than the parent index. Thus, no circular references are created between the tree nodes. Of course that the first node will have no parent since it is the root node. The variable parameters are: i) the size of the array "n", which we limited to a maximum of 18 nodes; ii) the name of the nodes, which are continuous letters from A to Z; iii) the index values in the parent array; iv) the maximum number of consecutive equal parents in the array, which will reflect the tree degree, limited to a maximum of n/3.

The second generic exercise is about inserting keys in a binary tree. We need to generate a set of random integer keys which will allow the creation of a balanced binary tree. In order to achieve this goal one simple solution is to define three integer partitions from where an equal number of keys will be selected. The variable parameters are: i) the number of keys in the tree, which we limit to a maximum of 12; ii) the extent of the partitions, where we set the value of 30. Thus, the keys will be provided from 0 to 29, 30 to 59 and from 60 to 89; iii) the three extraction probabilities from the partitions, which we consider to be all equal to 1/3 with the constraint that the first key should be from the second partition. As an improvement to this exercise we could consider that the keys could be selected from a words file repository where all words are sorted alphabetically.

The third generic exercise deals with drawing a graph diagram based on a randomly generated adjacency matrix. The variable parameters are: i) the size of the adjacency matrix having 6 rows and 6 columns; ii) the elements which can be 0 or 1, except the main diagonal which is all 0; iii) the grade of the graph; iv) the name of the nodes which are random alphabet big caps letters. After the graph diagram is drawn then the grade of the graph is asked. The two answers can be easily verified in an automatic manner when the assessment is performed on a computer. The graph diagram can be created using a diagram editor, and the node names help us check the correct links between the nodes. The graph grade is calculable out of the generated adjacency matrix.

The fourth generic exercise deals with node graph searches: i) depth-first search and ii) breadth-first search. The matrix generation algorithm is the same. The variable parameter is the starting node for both searches.

The fifth generic exercise is about determining the minimum coverage tree in a weighted graph. The graph generation algorithm is the same as the previous one except the values which are not only 0 or 1 but from 0 to 100. In order to have a balanced number distribution the values are generated with the formula:
rand()\%2 ? 0 : rand()\%100.

3 Assessment Items for Operating Systems Discipline

The proposed exercises will test the writing of some the Unix [13] commands: i) to create 3 directories with different random names; ii) to enter one of the three directories; iii) to create and edit a few files with writing some lines in them; iv) to copy a randomly selected file in some random target directory; v) to move some files with a random extension specification into a random target directory; vi) to assess the size of a file with a random name; vii) to change the access rights of an existing random file with a random set of given rights; viii) to delete recursively a given folder. These assessment items can be automatically evaluated by a simple parsing for white space eliminations. The presented assessment items can be also reused in the context of system calls where the student must write programs to fulfill the generated tasks. The answer program checking involves a more complex parsing and pattern recognition.

4 Assessment Items for Fundamentals of Programming Languages Discipline

For the Fundamentals of Programming Languages discipline we created four assessment items dedicated to its Lisp [11,16] laboratory. The first exercise item generates a multilevel list based on random dictionary words where the student has to apply the CAR and CDR primitives in order to extract the first occurrence of a certain letter from that word. The variable parameters in this assessment item are: i) the word selected randomly from a dictionary, which has a certain length; ii) the extracted letter which will be selected randomly from the second half of the

word. This exercise is more complex since we have to generate a tree where the inner nodes are non-important marked by stars * and the leaf nodes must contain, in order, the words letters. Such a generated result is presented in Figure 1.

```
(((P)(O))((((((L I)(T))(E))(H))(N))((I C)(A))))
```

Fig. 1. Generated Multi-Level List

A different assessment item was designed in order to stimulate student creativity. The exercise generates randomly a multilevel list and the student has to write a Lisp expression creating that list and using three primitives i) append; ii) list; iii) reverse. The word proposed for the exercise is (A B) expressed as a list. The variant parameters are: i) the word used in the exercise; ii) the number of nodes in the tree - equivalent with the multilevel list; iii) the degree of the tree; iv) the height of the tree. In figure 2 we present an example of the generated multilevel list based on the word (A B). Unless the word is not a palindrome the response expression is unique for each tree and thus can be assessed automatically.

```
((B A (A B) (A B A B) (A B) (A B A B)) ((A B) (B A B A)))
```

Fig. 2. Generated Multi-Level Word List

In Figure 3 we can see a generated simple expression based on a binary tree containing arithmetical operators and one letter identifier operands, implemented by an array with left child at 2*i and right child at 2*i+1 if the parent is at index i. These expressions must be implemented as Lisp functions by students. The variable parameters are: i) the number of operators, we set between 4 and 6; ii) the name of the identifier operands.

```
F3=((n + g) - (p + r))
```

Fig. 3. Generated Simple Expressions

For the complex expression we use the same generation idea but the number of operators will be between 7 and 12. In Figure 4 we can see a generated complex expression.

```
F4=((((g * p) * (q - b)) * ((g - i) - x)) +
       ((e * x) - (j * e)))
```

Fig. 4. Generated Complex Expressions

5 Assessment Items for Compiling Techniques Discipline

For the Compiling Techniques [2] discipline we designed one complex exercise which has the goal of generating randomly a variable grammar made of a lexical analyzer and a syntactical analyzer having a limited difficulty level. We designed a generic program structure described by a grammar having a lexical analyzer which contains: i) the identifier rule with different set of letters, digits and name; ii) keywords for starting and ending program blocks having fixed semantics, but variant synonym names; iii) separators selectable randomly from a given set; iv) operators selectable randomly from a given list; v) delimiters selectable randomly from a given set; vi) integer constants in different forms; vii) real constants in different forms.

```
letter ::= a..z | A..Z
dig ::= 0..7
operator ::= +
separator ::= ,
del1 ::= [
del2 ::= ]
identifier ::= (letter|dig)*
int_literal ::= dig+
nr_real ::= dig+.dig+
start_keyword ::= ON
stop_keyword ::= OFF
```

Fig. 5. Generated Lexical Rules

In the syntactical analyzer rule set we created: i) left-recursive expressions with randomly selected operators and operands, the operands can randomly be identifiers or integer constants or real constants; ii) instruction lists; iii) different instructions like assignments, calls etc; iv) starting grammar rule built with random keywords. An example of a generated set of syntactic rules is presented in Figure 6.

```
programs ::= programs start_keyword ListaInstr stop_keyword
  separator | start_keyword stop_keyword
InstrList ::= InstrList Instr | Instr
Instr ::= identifier := E | identifier del1 E del2 := E
E ::= E operator int_literal | E operator identifier
  | int_literal | identifier
```

Fig. 6. Generated Syntactic Rules

The skeleton of our compiler is quite simple but following these ideas we can build a larger compiler skeleton. Briefly, the variable parameters are: i) the names for the lexical and syntactical rules; ii) the right-hand side of the lexical rules with

variations for each token class; iii) the randomly included subtrees in the right-hand side of syntactical rules.

6 Prototype Implementation

The prototype is implemented in C and has two versions: i) the first one generating HTML code so the components behave as CGIs; ii) the second one generating LaTex code for translating in PDF format ready to print. The former result is good for online posting of free exercises, while the latter is good for written exams. Regarding the balance between the server and client side we can mention that the current implementation based on CGIs runs on the server side. We consider that is not a very difficult task to translate the code into JavaScript [12] or Flash ActionScript [1] which runs on the client side in order to enable a better graphical representation.
The action results can be memorized into the Learning Record Store (LRS) using the Experience API (xAPI) [14].

7 Related Works

According to [6] a GLO is "an articulated and executable learning design that produces a class of learning objects". Our approach adheres fully to these ideas.

In [4] are presented the design principles for creating dynamic and reusable LOs. The principles are based on a set of distilled ideas from pedagogy and software engineering. The case study is made on a Java learning discipline. With our approach we showed that GLOs can be used for several other computer science disciplines to a certain extent.

In [5] are presented design and development tools for the GLOs as second generation learning objects underpinning pedagogical patters. In our approach we reuse pedagogical patterns in a competence oriented learning and assessment.

In [7] they consider that GLOs are generic and reusable LOs from which specific content can be generated on demand. GLOs are characterized by variability which can be modeled using feature diagrams and they also need specification languages, parameterization languages, metaprogramming techniques for generation. In our approach the parameters are expressed through program variables, their values are set by random values within a certain range, so LO instantiation is automatic.

In [8] GLO generated LO sequences by metaprogramming are expressed using sequence feature diagrams. In our approach the metamodel is not explicit but it is embedded into the prototype modules code.

8 Conclusions and Future Work

In this paper we presented five modules containing generic learning assessment items dedicated to a set of disciplines that we consider belonging the core for

computer programming. In our approach we identified a number of problems that must be considered challenges for future work.

The assessment items test only the good understanding of the data structure or algorithm functioning which are essential for programming but not programming itself with that data structure. The generative assessment items seem to cover only a small part of the content a student must know. The part which is not covered involves the application of the data structures and algorithms in industrial strength programs.

Regarding the Operating System discipline we consider the following challenges: i) to generate script specifications to work with files and processes and to combine safely the possible operations; ii) to assess automatically the correctness of the written scripts. Using repeatedly the same generative assessment items it may transform the evaluation into a tedious activity.

The items are quite complex in creation and implementation, because they require programming knowledge for the values generation so the authors must have programmer skills. Another challenge deriving from this idea is to simplify the implementation of such GLO items by using specialized templates or generative wizards constructing the output step by step.

As future work we consider integrating the designed GLOs into a Learning Management System like Moodle [9]. Thus, the GLOs will be available to a larger number of tutors and implicitly students.

For the motivational part we can think of integration with social networks. Thus, the learning activity result can be posted online and get support and approval by the other members of the community. They may try themselves some interesting e-learning topics.

Another future work is related to gamification, namely to use gaming mechanics to transform some GLOs into games. DSA modules are more likely to be able to be gamified since it involves lots of diagrams and interactions between nodes.

In order to support learning and training together with evaluation, some assessment items could be equipped with assistance and feedback in order to be reused as training items.

Acknowledgements

This paper is supported by the Sectorial Operational Programme Human Resources development (SOP HRD), financed from the European Social Fund and by the Romanian Government under the project number POSDRU/159/1.5/S/134378 initiated in 2014.

References

1. Adobe Systems. Flash ActionScript. http://www.adobe.com, 1998.
2. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman – Compilers, Principles, Techniques and Tools, Second Edition, Pearson Education, 2006.

3. Diana Andone, Jon Dron, Lyn Pemberton, and Chris Boyne. E-learning environments for digitally-minded students. Journal of Interactive Learning Research, 18(1):41–53, 2007.

4. Tom Boyle. Design principles for authoring dynamic, reusable learning objects. Australian Journal of Educational Technology, 18(1):46–58, 2003.

5. Tom Boyle. The design and development of second generation learning objects. In Proceedings of World Conference on Educational Multimedia, Hypermedia & Telecommunications, 2006.

6. Centre for Excellence for the design, development and use of learning objects. Reusable learning objects cetl - rlo-cetl. http://www.rlo-cetl.ac.uk/whatwedo/glos/glodevelopment.php, 2014.

7. Robertas Damaeviius and Vytautastuikys. On the technological aspects of generative learning object development. Lecture Notes in Computer Science, 5090:337–348, 2008.

8. Robertas Damaeviius and Vytautas tuikys. Using sequence feature diagrams and metaprogramming techniques. In 2009 Ninth IEEE International Conference on Advanced Learning Technologies, 2009.

9. Martin Dougiamas. Modular object-oriented dynamic learning environment (Moodle). http://www.moodle.org, 2002.

10. Donald E. Knuth - Art of Computer Programming, Volume 3: Sorting and Searching (2nd Edition), 800 pages, Addison-Wesley Professional, 2 edition (May 4, 1998), ISBN-10: 0201896850, ISBN-13: 978-0201896855, 1998.

11. John McCarthy. History of Lisp. History Of Programming Languages, Artificial Intelligence Laboratory Stanford University, February 1979.

12. Mozilla Foundation. JavaScript. https://www.mozilla.org, 2011.

13. Dennis Ritchie, Ken Thompson. The UNIX Time-Sharing System. Communications of ACM, vol. 17, no. 7, pp. 365--375, July 1974.

14. Rustici Software. Tin can api. http://tincanapi.com/, 2014.

15. Robert Sedgewick, Philippe Flajolet - An Introduction to the Analysis of Algorithms (2nd Edition). 592 pages, Addison-Wesley Professional; 2 edition (January 28, 2013), ISBN-10: 032190575X, ISBN-13: 978-0321905758, 2013.

16. David S. Touretzky. Common LISP: A Gentle Introduction to Symbolic Computation (Dover Books on Engineering), January 24, 2013.