Learners Programming Language a Helping System for Introductory Programming Courses

MUHAMMAD SHUMAIL NAVEED*, MUHAMMAD SARIM*, AND KAMRAN AHSAN*

RECEIVED ON 27.03.2015 ACCEPTED ON 16.09.2015

ABSTRACT

Programming is the core of computer science and due to this momentousness a special care is taken in designing the curriculum of programming courses. A substantial work has been conducted on the definition of programming courses, yet the introductory programming courses are still facing high attrition, low retention and lack of motivation. This paper introduced a tiny pre-programming language called LPL (Learners Programming Language) as a ZPL (Zeroth Programming Language) to illuminate novice students about elementary concepts of introductory programming before introducing the first imperative programming course. The overall objective and design philosophy of LPL is based on a hypothesis that the soft introduction of a simple and paradigm specific textual programming can increase the motivation level of novice students and reduce the congenital complexities and hardness of the first programming course and eventually improve the retention rate and may be fruitful in reducing the dropout/failure level. LPL also generates the equivalent high level programs from user source program and eventually very fruitful in understanding the syntax of introductory programming languages. To overcome the inherent complexities of unusual and rigid syntax of introductory programming languages, the LPL provide elementary programming concepts in the form of algorithmic and plain natural language based computational statements. The initial results obtained after the introduction of LPL are very encouraging in motivating novice students and improving the retention rate.

Key Words: Programming, Introductory Programming Courses, Imperative Programming, Student Dropouts.

1. INTRODUCTION

Programming skill is an expected outcome of computer science students [1]; however, it is hard to learn programming as it fundamentally entails to think and verbalize in a way that is unnatural and usually queer for the beginner. In [2], Winslow claims that approximately ten years are required to turn the beginner into an expert.

The majority of students face difficulties in comprehending the programming which results in high failure and dropouts in introductory programming courses. In [3-4], the failure/dropout rates of around 25% are reported in introductory programming courses, similarly 27.8% is reported in [5], and dropout of 40-60% is reported in [6].

* Department of Computer Science, Federal Urdu University of Arts, Science & Technology, Karachi.

Akin to the impact of our native language on our cogitation and apprehension, the first programming language has a strong impact on our thoughts [7], and similarly the programming language chosen for the introductory programming course is an essential element in the progress of students [8].

The paradigm of the first programming course is one of a pivotal factor in its success. Among several programming paradigms, the imperative (also called procedural) paradigm is highly acknowledged for introductory courses [9-12], yet its various features are quite hard for novice students. For instance, in [13], the sequence, assignment and iteration are cited as the difficult concepts. Whereas in [14], the control structures, pointers, arrays and recursion are classified as difficult concepts for beginners. Gobil, et. al. argued that beginners face problems in recognizing the valid selection structure for the problems [15].

Virtually there is a series of factors for high attrition in computer programming courses, but lack of previous knowledge of programming is an important reason that affects the completion of computer science degree [16-18]. Conversely the novice students with previous knowledge of programming have better performance [19-20], and based on this thesis, it can be inferred that basic concepts of the first programming course can become easier if novice students have prior knowledge of programming. In this paper we have introduced a LPL which is based on a thesis that induction of small and plain language before a first programming course can appreciably increase retention levels of novice students and decrease attrition rate by providing them elementary acquaintance of imperative programming.

2. PREVIOUS WORK

The notion of pre-programming language as a precursor of introductory programming courses is not very new; several programming systems have been formalized to support the first programming courses. McIver et al. introduced a language called GRAIL and inducted before the first course of programming [21]. GRAIL helps the novice students in comprehending the imperative programming concepts.

In [22], a course is introduced to help novice students who have no prior programming knowledge. The course covers the elementary topics like sequence, selection structures, operators, arrays and functions.

In another landmark study [23], the effectiveness of CS0 with Scratch [24] is studied. The course encompasses the sequence, variables, arrays, selection, repetition, and basic objects. The course significantly reduced the attrition rate.

In another study [25], a course called Computational Thinking is described to help novice students and found very effective in increasing the problem solving abilities of beginners.

To trounce the difficulties of introductory programming courses the MindStorms robots based course is introduced to encourage the beginners to focus on algorithmic problems solving before introducing the Java [26].

Many other courses and pre-programming languages have been developed to support introductory programming courses. For instance [27-33] for more detail.

3. LEARNERS PROGRAMMING LANGUAGE

The LPL is a tiny precursor programming language designed to help the novice students by providing elementary concepts which are helpful in comprehending first introductory imperative programming course CS1 (Computer Science). The LPL is specifically designed to support introductory imperative programming courses;

however, the notion of LPL can be applied to support introductory programming courses of other paradigms [34-35]. The principal reason for the selection of the imperative paradigm for the LPL is its wide application usage. In the curriculum of CS undergraduate degree programs the majority of introductory programming courses (particularly the first programming course) are typically based on imperative paradigm. The landmark study [12] on introductory programming courses in the universities of Australia and New Zealand report that imperative paradigm is the preferred paradigm as illustrated in Fig. 1.

In Pakistan there are 164 universities/degree awarding institutions [36], and according to accessible information, more than 100 institutions are offering the pure undergraduate CS programs and nearly all are employing imperative paradigm for a first programming course.

3.1 Design Theory and Motivation

The overriding theory of LPL is to facilitate the learning of introductory programming in a way to help the novice students in comprehending first imperative programming course. Principally it is merely not developed to ease the programming - if that was the objective; the graphical tools would be ample to provide simple drag-and-drop facilities to construct the programs. Fundamentally the ZPL course and first programming course are radically different, since the former is aimed is to introduce the elementary programming concepts, whereas the later is the actual platform to explore the actual concepts to beginners and therefore LPL is primarily developed for introducing elementary programming, rather than for doing programming.

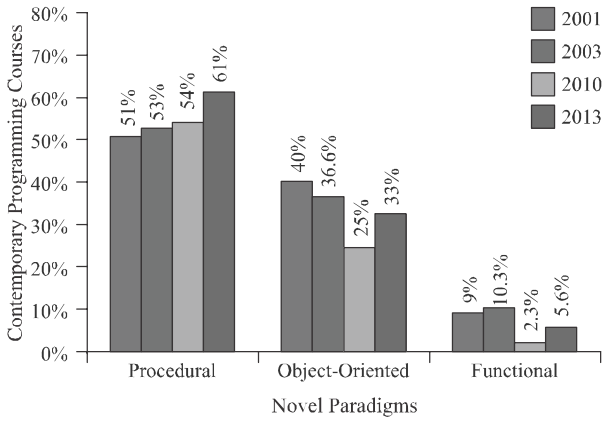

FIG. 1. TRENDS IN PARADIGM TAUGHT [12]

3.2 Prime Features

The LPL requires no explicit perquisite. The overall syntax of LPL is extremely simple and only a small number of features are included. Currently it includes primitive data types (character, integer, float), string, literals, variable declaration, I/O, linear array of primitive data types, selection structure (single, double, multiple), control structures (counter and logical pre-test) and single line comments.

Several studies found that unfamiliar and unusual syntax of programming languages makes it more confusing and difficult [37]. The structure and denotation of LPL are very simple and obvious. All the available features are presented in algorithmic/natural language based computational statements, and therefore, it is far simpler to use LPL and understand the elementary concepts.

The LPL's translator converts the input source into the equivalent high level code in C and C++ and therefore very helpful in understanding rigid syntax of the base language of introductory programming course.

All the statements of LPL programs are enclosed in functions. Like conventional imperative programming languages the *main function* is the entry point of the program. Declaration of main function may take several forms. One of them is:

start of the main program

…

end of main program

The statements within a function are virtually divided into two sections. The first section contains the declaration of variables, whereas the second section includes actual statements. The declaration of main section/function in most of the imperative programming languages like the C is extremely difficult and generally requires the knowledge of return value and arguments; however, at elementary level it is unjustifiable to introduce these peculiarities.

The LPL allows the declaration of variables and all variables are strongly typed and require proper declaration. The declaration statement may take several forms, however, each of the possible forms is prefixed with the word "create" (or some equivalent):

```
declare a float type variable named fee
create an integer variable named grade
define float amount
```

At first glance, the general syntax for the declaration of variables seems too verbose, however, in reality the descriptive style of programming and variable declaration help the novice students in understanding the fundamental of programming. As a convenience and flexibility the LPL also allows the declaration of variables in a concise form, like "*define float amount"*.

The traditional symbol (=) is used for assignment, and likely to add dot-equal notation for assignment.

The complex I/O is one of the reasons that make programming language more hard and complex. The I/O mechanism in most of the programming language is extremely difficult, however, it should be simple and straightforward [38].

Simple and straightforward statements are used in LPL for console I/O. The input statement may take one of the following forms:

```
input in name
take input in name
```

Similarly the output statement may take several forms:

```
print the value of grade
show grade
display age
```

According to Dale, the control structures are one of the difficult topic for novice students [14], and due to this reason a special care is taken while defining the control structures. The iteration structure in LPL may take several forms; however, each of the possible forms is prefixed by the word "repeat" (or some equivalent):

```
repeat the statements 500 times
        ...
end of loop
```

or

```
repeat as long as age < 50
        ...
end of loop
```

A very comprehensive study, reported in [39], revealed that students face difficulties in identifying the right selection structure for the given problems. The selection structure in LPL, may take the following forms, but each of the possible forms is prefixed with the word "execute" (or some equivalent):

```
execute if age > 50
        ...
end of if
```

or

```
execute on the condition that number <> 100
                ...
end of condition
```

Single line comment stared with "#" is allowed in LPL, whereas the references and pointers are not available in LPL.

In LPL each syntactic structure (other than the expression) is started with the unique word, and this feature obviously helps the novice students in understanding, differentiating and implementing programs and their pertinent components.

3.3 Language Design

The LPL is developed by using the universal notions of the theory of automata and formal language. At definition level, the LPL is divided into three sections: lexical specification, structural specification and semantic specification.

During the definition of lexical specification, the lexical units (keywords, literals, identifiers, operators and punctuators) are defined. The universally accepted mathematical system called the regular expression [40] is used for definition of lexical specification. The defined lexical specification for the LPL is highly straightforward and amenable.

Consider the regular expression for the 'identifier' LPL:

letter '! A | B | C | … | Z | a | b | … | z
digit '! 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
identifier '! letter (letter | digit)*

With the above regular expression, it is possible to create any valid identifier in LPL.

As an illustration, consider the regular expression for integer constant in LPL:

constantliteral '! (+ | - | ε) digit digit*
digit '! 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

The above regular expression can only define the integer constant; however, its extended form can define the floating value constant:

floatliteral '! (+ | - | ε) digits . digits (e | E) (+ | - | ε) digits
digits '! digit digit*
digit '! 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

The regular expressions for the other lexical units like keywords and operators are highly straightforward and defined by concatenating the relevant symbols in a specific order.

After the specification of lexical units the structural specification of LPL is defined by using the context free grammar [41]. The context free grammar of LPL is somewhat massive, and the reason behind this bigness is the definition of several semantic specifications at the structure level.

Consider a segment of the context free grammar for a definition of the function header:

<header> '! VERBSTART <mphrase>
<mphrase> '! PREPOSITIONOF <article>
<fname> PROGRAMSYMBOL | ε
<article> '! ARTICLEA | ARTICLETHE | ε
<fname> '! FUNCTIONNAME | ε

As an illustration consider the following segment of context free of the output statement:

<statement> '! VERBOUTPUT <article> <parameter>
<article> '! ARTICLETHE | ARTICLEA | | ε
<parameter> '! CONSTANT | IDENTIFIER | VALUE <trail>
<trail> '! CONSTANT | PREPOSITION OF IDENTIFIER

The semantic specification is the last specification of the LPL. During this level the type checking information is defined by using the attribute grammar [42]. For definition of attribute grammar the semantic attributes are attached with the actual context free grammar.

After the definition of LPL a small development environment as shown in Fig. 2 is developed to create, edit and translate the LPL programs into the high level codes of C and C++. The environment is developed in C# 3.0 by using the Visual Studio 2008.

The development environment of LPL comprises of two sections (code section and debug section) as shown in Fig. 2.

The environment includes a comprehensive help document that provides all the relevant information. The help documentation is created with Dr. Explain. The environment also includes the documentation about the detail of the errors may encounter during the translation of the program. The documentation about errors is also created with Dr. Explain. For better error handling, a unique number is assigned to each possible error in LPL.

The environment includes a tiny translator that analyzes the LPL program and generates equivalent codes of C and C++. The translator of LPL highly mimics the structure of conventional compilers, and quite similar to transpilers. The LPL translator comprises of four main phases and the symbol table and the error handler are also the important components of LPL's translator.

Lexical analyzer is the first phase of translator and responsible to recognize all lexical elements of the program and identify their categories by generating the equivalent tokens. The finite state automata [43] are developed from the regular expressions for the recognition of lexemes. The generated tokens are passed to syntax analyzer which is the second phase of the translator. In LPL's syntax analyzer the recursive descent parsing is used [44]. In recursive descent parser, a separate method (procedure) is developed for every nonterminal of context free grammar. To simplify the construction of recursive descent parsing the actual context free grammar is initially transformed into the EBNF (Extended Backus-Naur Form). For error handling the panic mode recovery is used. The recursive descent parser verifies the syntax of the program and generates the implicit syntax tree. The implicit syntax is used by the semantic analyzer which is a third phase of translator that renovates the syntax tree by attaching and evaluating semantic attributes described during the specification of attribute grammar. The attribute are evaluated with tree traversing algorithms. In this way the semantic analyzer also performed the duty of intermediate code generator, otherwise it is a separate phase of conventional compilers. Once the parsing and successful evaluation of attributes are performed the target code generator which is a last phase of LPL's translator is invoked that translate the program into high level code in C and C++ as shown in Fig 3.

The target code generator simply performs the ad-hoc mapping by reading the syntactic structure of the input program and generates the equivalent programs. Since

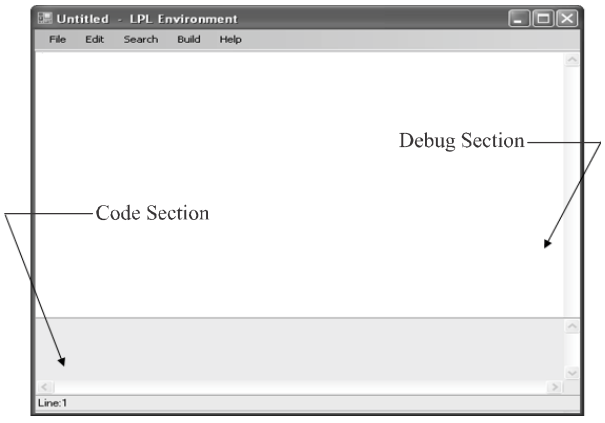

FIG. 2. LPL ENVIRONMENT

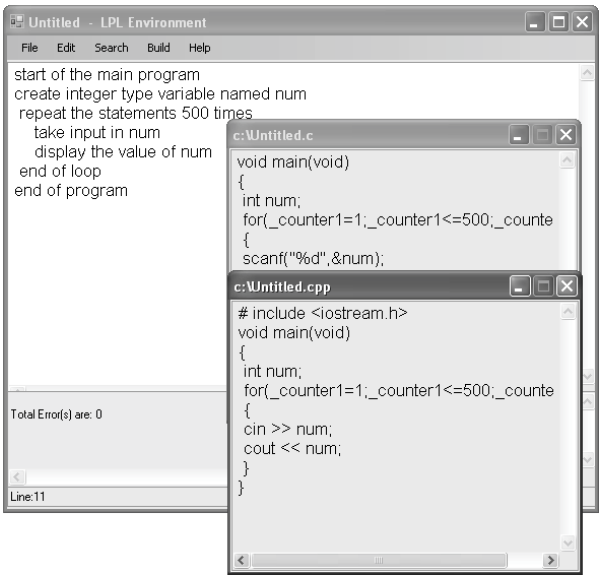

FIG. 3. LPL ENVIRONMENT WITH OUTPUT

the semantic analysis in LPL's translator is quite simple and no optimization is required during translation so the recursive descent parser is defined as a main program which call subprograms for semantic analysis and target code generation at suitable points as suggested by Wilhelm and Maurer [40].

4. PRELIMINARY ANALYSIS OF LPL

The LPL is an experimental language designed on a hypothesis that the induction of a small, simple and paradigm specific textual preprogramming language can help the students in comprehending first imperative programming course and resultantly increases the retention level and may also reduce the failure/dropout rates. In order to verify our theory a small study is conducted. During study 96 students of the undergraduate computer science program are randomly selected and categorized in three groups (A, B, C). As a part of our study the Python is selected as a ZPL for group A, whereas the LPL is selected for group B, however, no preprogramming is selected for group C.

For CS0 there are many popular languages. Fig. 4 illustrates the popular languages choice used in CS0.

According to the statistics given in Fig. 4, the Alice is the most popular language for CS0, whereas the Python and VB are the second popular languages in CS0. However,

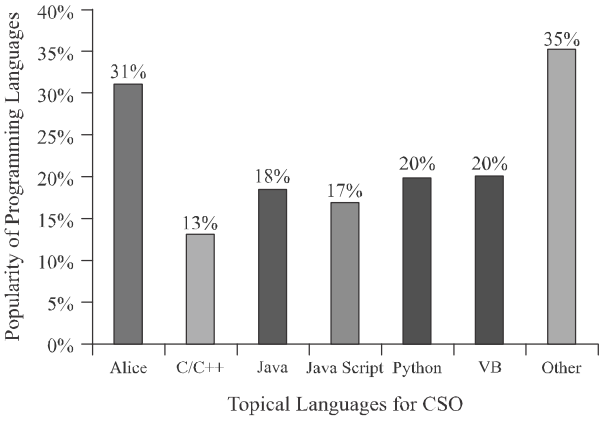

FIG. 4. POPULAR LANGUAGES USED IN CS0 [39]

due to several genuine reasons the Alice is not selected for comparing the effectiveness of LPL. The Alice is used for teaching the object oriented programming [45-46], whereas the LPL is developed for introductory imperative programming courses.

During our study the same number of lectures was designated for the both groups (A and B) for introducing their respective ZPLs. After the induction of ZPL the introductory programming course (CS1) with the C language is offered to all the three groups.

After the completion of course, the students from every group were interviewed and asked "whether they are willing to take another programming course in the next semester". The feedback received from the students is shown in Fig. 5.

The retention rate of the students in group A is quite higher from the students in group C but lower than the students in group B. The students from group A were asked about the reason of their less retention in the next programming course. Most of the students respond that the language used in CS0 was very attractive, but they faced problems when switching to the actual programming language of CS1, however, one student A plaint the shortage of time and the massiveness of language used in CS0. The retention level of the students in group B who studied the LPL as a ZPL is quite higher

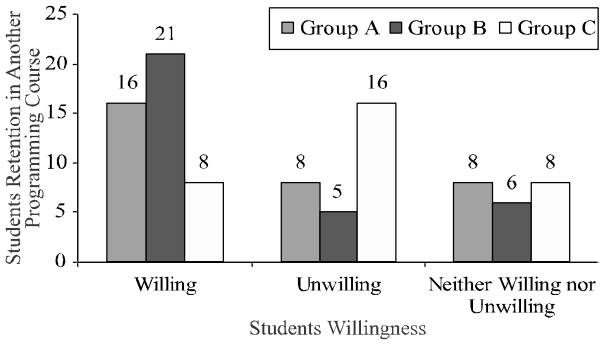

FIG. 5. STUDENTS WILLINGNESS IN THE NEXT PROGRAMMING COURSE

than the other class. The students from group B were asked about the reason of retention in the next programming course. Most of the students respond that the simple and understandable statements help them in understanding concepts, and with the feature of the high level code generation, they easily understand the various concepts and syntax of the actual programming language. Two students from group B who were not willing in the next programming course demand the detailed external document for the LPL. While interviewing the students of the group C, most of the students lament the hardness of programming and express their difficulties in abstracting and implementing the program ideas. Several students also complained the rigid and unfamiliar syntax of the programming language.

The students from all the groups are also being internally evaluated with pen and paper exam. The summary is shown in Fig. 6.

The use of LPL as a ZPL is also useful in improving the failure rate. In fact the radical improvement of the failure / drop rate is a complex subject and virtually based on several parameters. The intervention of any ZPL should not be considered as a complete panacea as it cannot improve the failure rate in the exponential order.

The initial evaluation of LPL and the analysis of its design philosophy signify that LPL can be a very helpful

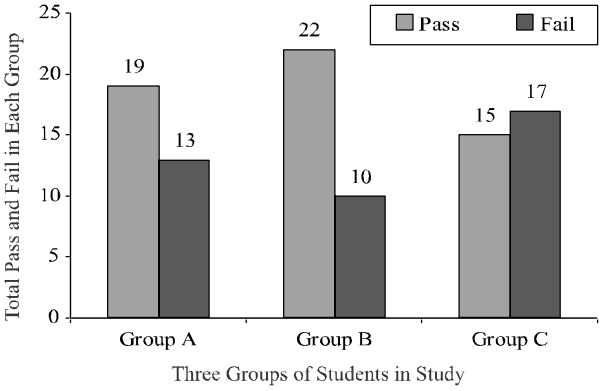

FIG. 6. RESULT OF EVALUATION

precursor for introductory imperative programming courses. There are tens of programming languages available for preprogramming and most of them are developed by multinational companies, but the majority of them is developed as a blanket solution for introductory programming courses whereas the LPL is a paradigm specific tool and only aims to support the introductory imperative programming courses. Almost all the popular programming languages (like the Python and Visual Basic) for CS0 require a complete course in a semester before the first programming course, whereas few many lectures of LPL are sufficient before commencing the first programming course. Almost all the imperative languages are primitively text oriented and LPL is also text oriented so it is more effective for the imperative languages.

Alice is one of a powerful language and generally used for teaching object oriented programming. In [47], Powers, et. al. analyzed the advantages and disadvantages of using Alice in CS0 and report that Alice increased the retention and confidence, and the lack of syntax error may increase the confidence of students, yet it can be problematic when student switches to C++ or Java. However, in LPL there is no issue of transition, since it is already text oriented and the target languages are also textual.

Blockly is a type of visual development kit that allows the quick development of new block-based visual programming languages [47]. Blockly is neither originated as a ZPL nor it supports any specific programming paradigm, however it permits the educational games for comprehending the notion of condition and iteration structures, yet the elementary concepts of imperative programming like the data type and scope are almost impossible to comprehend with Blockly. Basically the Google Blockly is a web-oriented, visual graphical programming editor that allows the development of applications by joining the blocks without the knowledge of programming [48], whereas the LPL is dedicatedly

developed as a ZPL and supports the imperative paradigm. Furthermore, it encourages and engages the students to learn the programming from the very beginning. The Blockly isolates the user from syntax errors, whereas the LPL aims to provide a gentle system for reducing the syntax error. Apparently it is not unfair to affirm that Blockly simplify the hardness of contemporary programming by providing the drag-and-drop facility, whilst the LPL targets to simplify the introductory programming by providing simple and understandable notations.

Stencyl is a platform developed by StencylWorks for the creation of video games [49]. It allows a graphical editor to create games and port them to different formats. In Stencyl the development of games is based on the description of actors and scenes. The environment of Stencyl is very innovative yet it requires algorithmic logic and programming knowledge [50], however for ZPL no previous knowledge of programming is principally required, moreover no explicit feature of Stencyl is directly supportive in comprehending the elementary concepts of imperative programming and therefore due to these reasons the Stencyl can't be considered a viable option for the effective precursor of introductory imperative programming courses.

Scratch is an environment and a media-rich programming language and developed by the Lifelong Kindergarten group with the UCLA Graduate School of Education and Information Studies [51]. It is generally used in the K-12 system. Scratch is usually used for teaching object oriented programming where the LPL is developed for imperative paradigm. Like other visual languages, it is free from the distraction of syntax, and unlike the textual languages, the use Scratch provides no explicit experience of program debugging. The procedure and recursion, which are the essential elements of imperative programming are missing in Scratch and the support for data structures is also weak [52].

Visual languages simplify the programming by providing the drag-and-drop facilities and prevent the syntax errors, but the students never confronted with program coding and never gain any practical experience of coding and debugging. In fact the visual languages are useful and simple yet several studies indicate that graphical languages are only useful for certain purposes [53]. In [54], it is argued that constructs based on control flow are linear (with some exceptions), and therefore, easily symbolized with textual language, and the visual languages are more suitable for representing the constructs which are based on data flow. The graphical representation was better than textual pseudo code when the operation entails finding flow of control, but not for identifying the high level associations [55], and as far as the program comprehension is concerned the graphical representations are not superior to textual representation and sometimes seriously worse [56].

5. CONCLUSION

LPL is a ZPL that helps the novice students in comprehending the introductory programming course. LPL comprises of simple and understandable statements and generates the equivalent C and C++ source program from the user source programs. LPL is initially applied to a small group of students and found very positive in increasing the retention level and reducing the failure rate. As a next step we are excogitating to apply LPL to a large group of students with the aim of quantifying its successfulness and efficacy in increasing the motivation level, retention rate and decreasing the attrition rate.

ACKNOWLEDGEMENTS

The support provided by the Department of Computer Science, Federal Urdu University of Arts, Science & Technology, Karachi, Pakistan, is gratefully acknowledged. Authors would also like to thank Abdul Basit and Kashif Raffat, for their valuable criticisms on learners programming language. In addition, authors would like to thank all of the students who were willing to share their time with us in our studies.

REFERENCES

[1] McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y.B., Laxer, C., Thomas, L., Utting I., and Wilusz, T., "A Multi-National, Multi-Institutional Study of Assessment of Programming Skills of First-year CS Students", ACM SIGCSE Bulletin, Volume 33, No. 4, pp.125-180, 2001.

[2] Winslow, L.E., "Programming Pedagogy: A Psychological Overview", ACM SIGCSE Bulletin, Volume 28, No. 3, pp. 17-22, 1996.

[3] Herrmann, N., Popyack, J.L., Char, B., Zoski, P., Cera, C.D., Lass, R.N., and Nanjappa, A., "Redesigning Introductory Computer Programming Using Multi-Level Online Modules for a Mixed Audience", ACM SIGCSE Bulletin, Volume 35, No. 1, pp. 196-200, 2003.

[4] Nagappan, N., Williams, L., Ferzli, M., Wiebe, E., Yang, K., Miller, C., and Balik, S., "Improving the CS1 Experience with Pair Programming", ACM SIGCSE Bulletin, Volume 35, No. 1, pp. 359-362, 2003.

[5] Rich, L., Perry, H., and Guzdial, M., "A CS1 Course Designed to Address Interests of Women", ACM SIGCSE Bulletin, Volume 36, No. 1, pp. 190-194, 2004.

[6] Sloan, R.H., and Troy, P., "CS 0.5: A Better Approach to Introductory Computer Science for Majors", ACM SIGCSE Bulletin, Volume 40, No. 1, pp. 271-275, 2008.

[7] Wexelblat, R.L., "The Consequences of One's First Programming Language", Proceedings of 3rd ACM SIGSMALL Symposium and the First SIGPC Symposium on Small Systems, pp. 52-55, USA, 1980.

[8] Luker, P.A., "Never Mind the Language, What About the Paradigm?", ACM SIGCSE Bulletin, Volume 21, No. 1, pp. 252-256, 1989.

[9] deRaadt, M., Watson, R., and Toleman, M., "Language Trends in Introductory Programming Courses", Proceedings of Informing Science and IT Education Conference, pp. 329-337, Ireland, 2002.

[10] deRaadt, M., Watson, R., and Toleman, M., "Introductory Programming: What's Happening Today and Will There be Any Students to Teach Tomorrow?", Proceedings of 6th Australasian Conference on Computing Education, Volume 30, pp. 277-282, New Zealand, 2004.

[11] Mason, R., Cooper, G., and deRaadt, M., "Trends in Introductory Programming Courses in Australian Universities: Languages, Environments and Pedagogy", Proceedings of 14th Australasian Conference on Computing Education, Volume 123, pp. 33-42, Australia, 2012.

[12] Mason, R., and Cooper, G., "Introductory Programming Courses in Australia and New Zealand in 2013 - Trends and Reasons", Proceedings of 16th Australasian Conference on Computing Education, pp. 139-147, New Zealand, 2014.

[13] Vujoševic-Janicic, M., and Tošic, D., "The Role of Programming Paradigm in the First Programming Courses", The Teaching of Mathematics, Volume XI, No. 2. pp. 63-83, 2008.

[14] Dale, N.B., "Most Difficult Topics in CS1: Results of an Online Survey of Educators", ACM SIGCSE Bulletin, Volume 38, No. 2, pp. 49-53, 2006.

[15] Gobil, A., Shukor, Z., and Mohtar, I.A., "Novice Difficulties in Selection Structure", International Conference on Electrical Engineering and Informatics, Volume 2, pp. 351-356, Malaysia, 2009.

[16] Davy, J.R., Audin, K., Barkham, M., and Joyner, C., "Student Well-being in a Computing Department", ACM SIGCSE Bulletin, Volume 32, No. 3, pp. 136-139, 2000.

[17] Hagan, D., and Markham, S., "Does It Help to Have Some Programming Experience Before Beginning a Computing Degree Program?", ACM SIGCSE Bulletin, Volume 32, No. 3, pp. 25-28, 2000.

[18] Morrison, M., and Newman, T.S., "A Study of the Impact of Student Background and Preparedness on Outcomes in CS I", ACM SIGCSE Bulletin, Volume 33, No. 1, pp. 179-183, 2001.

[19] Holden, E., and Weeden, E., "The Impact of Prior Experience in an Information Technology Programming Course Sequence", Proceedings of 4th Conference on Information Technology Curriculum, pp. 41-46, USA, 2003.

[20] Tafliovich, A., Campbell, J., and Petersen, A., "A Student Perspective on Prior Experience in CS1", Proceeding of 44th ACM Technical Symposium on Computer Science Education, pp. 239-244, USA, 2013.

[21] McIver, L., Linda, M., and Conway, D., "GRAIL: A Zeroth Programming Language", Proceedings of 7th International Conference on Computing in Education, pp. 43-50, The Netherlands, 1999.

[22] Panitz, M., Sung, K., and Rosenberg, R., "Game Programming in CS0: A Scaûolded Approach", Journal of Computing Sciences in Colleges, Volume 26, No. 1, pp. 126-132, 2010.

[23] Rizvi, M., Humphries, T., Major, D., Lauzun, H., and Jones, M., "A new CS0 Course for At-Risk Majors", 24th IEEE-CS Conference on Software Engineering Education and Training, pp. 314-323, Hawaii, 2011.

[24] Meerbaum-Salant, O., Armoni, M., and Ben-Ari, M., "Learning Computer Science Concepts with Scratch", Computer Science Education, Volume 23, No. 3, pp. 239-264, 2013.

[25] Van Dyne, M., and Braun, J., "Effectiveness of a Computational Thinking (CS0) Course on Student Analytical Skills", Proceedings of 45th ACM Technical Symposium on Computer Science Education, pp. 133-138, USA, 2014.

[26] Ernest, J.C., Bowser, A.S., Ghule, S., Sudireddy, S., Porter, J.P., Talbert, D.A., and Kosa, M.J., "Weathering MindStorms with Drizzle and DIODE in CS0", ACM SIGCSE Bulletin, Volume 37, No. 3, pp. 353-353, 2005.

[27] Dierbach, C., Taylor, B., Zhou, H., and Zimand, I., "Experiences with a CS0 Course Targeted for CS1 Success", ACM SIGCSE Bulletin, Volume 37, No. 1, pp. 317-320, 2005.

[28] Moskal, B., Lurie, D., and Cooper, S., "Evaluating the Effectiveness of a New Instructional Approach", ACM SIGCSE Bulletin, Volume 36, No. 1, pp. 75-79, 2004.

[29] Cooper, S., Dann, W., and Pausch, R., "Alice: A 3D Tool for Introductory Programming Concepts", Journal of Computing Sciences in Colleges", Volume 15, No, 5, pp. 107-116, 2000.

[30] Powers, K., Ecott, S., and Hirshfield, L.M., "Through the Looking Glass: Teaching CS0 with Alice", ACM SIGCSE Bulletin, Volume 39, No. 1, pp. 213-217, 2007.

[31] Agarwal, K.K., and Agarwal, A., "Simply Python for CS0", Journal of Computing Sciences in Colleges, Volume 21, No. 4, pp. 162-170, 2006.

[32] Agarwal, K.K., Agarwal, A., and Celebi, M.E., "Python Puts a Squeeze on Java for CS0 and Beyond", Journal of Computing Sciences in Colleges, Volume 23, No. 6, pp. 49-57, 2008.

[33] Agarwal, K.K., Agarwal, A., and Fife, L., "Python and Visual Logic: A Good Combination for CS0", Journal of Computing Sciences in Colleges, Volume 27, No. 4, pp. 22-27, 2012.

[34] Sebesta, R.W., "Concepts of Programming Languages", Addison- Wesley, 10th Edition, USA, 2012.

[35] Louden, K.C., and Lambert, K.A., "Programming Languages: Principles and Practice", Cengage Learning, 3rd Edition, USA, 2011.

[36] http://www.hec.gov.pk/Ourinstitutes/pages/Default.aspx (last accessed: 2nd April 2015).

[37] Petrov, P.T., "New Evaluation of the Language C for Educational, Engineering and Scientific Purposes", International Scientific Conference, pp. 353-361, Bulgaria, 2010.

[38] McIver, L., "The Effect of Programming Language on Error Rates of Novice Programmers", Annual Workshop of the Psychology of Programming Interest Group, pp. 181-192, Italy, 2000.

[39] Davies, S., Polack-Wahl, J.A., and Anewalt, K., "A Snapshot of Current Practices in Teaching the Introductory Programming Sequence", 42nd ACM Technical Symposium on Computer Science Education, pp. 625-630, USA, 2011.

[40] Wilhelm, R., and Maurer, D., "Compiler Design", Addison Wesley, England, 1995.

[41] Martin, J.C., "Introduction to Languages and The Theory of Computation", The McGraw-Hill Companies, 4th Edition, USA, 2010.

[42] Louden, K.C., "Compiler Construction: Principles and Practice", PWS Publishing Company, USA, 1997.

[43] Linz, P., "An Introduction to Formal Languages and Automata", Jones & Bartlett Publishers, 3rd Edition, USA, 2000.

[44] Aho, V.A, Lam, M.S., Sethi, R., and Ullman, J.D., "Compilers: Principles, Techniques, and Tools", Addison Wesley, Boston, 2nd Edition, USA, 2006.

[45] Cooper, S., Dann, W., and Pausch, R., "Alice: A 3D Tool for Introductory Programming Concepts", Journal of Computing Sciences in Colleges, Volume 15, No. 5, pp. 107-116, 2000.

[46] Jeanette, S.R., "From Alice to Blue: A Transition to Java", Master Thesis, Robert Gordon University, 2009.

[47] Trower, J., and Gray, J., "Creating New Languages in Blockly: Two Case Studies in Media Computation and Robotics", Proceedings of 46th ACM Technical Symposium on Computer Science Education, pp. 677-77, USA, 2015.

[48] Yuanhong X., "Using Blockly to Create Simple Sensor & Actuator Based Applications on the Sensible Things Platform", Degree Project, Mid Sweden University, 2014.

[49] Stencyl, L., "Stencyl: Design Once, Play Anywhere", Available: http://www.stencyl.com/

[50] Valdez, E.R.N., Martínez, Ó.S., Bustelo, B.C.P.G., Lovelle, J.M.C., and Hernandez, G.I., "Gade4all: Developing Multi-platform Videogames based on Domain Specific Languages and Model Driven Engineering", International Journal of Artificial Intelligence and Interactive Multimedia, Volume 2, No. 2, pp. 33-42. 2013.

[51] Rizvi, M., Humphries, T., Major, D., Jones, M., and Lauzun, H., "A CS0 Course Using Scratch", Journal of Computing Sciences in Colleges, Volume 26, No. 3, pp. 19-27, 2011.

[52] Harvey, B., and Mönig, J., "Bringing 'No Ceiling' to Scratch: Can One Language Serve Kids and Computer Scientists?", Constructionism, pp. 1-10, 2010.

[53] Gilmore, D.J., and Smith, H.T., "An Investigation of the Utility of Flowcharts During Computer Program Debugging", International Journal of Man-Machine Studies, Volume 20, No. 4, pp. 357-372, 1984.

[54] Green, T.R., Petre, M., and Bellamy, R.K.E., "Comprehensibility of Visual and Textual Programs: A Test of Superlativism Against the 'Match-Mismatch' Conjecture", Empirical Studies of Programming, 4th Workshop, Ablex Publishing Corporation, pp. 121-146, Canada, 1991.

[55] Curtis, B., Sheppard, S.B., Kruesi-Bailey, E., Bailey, J., and Boehm-Davis, D.A., "Experimental Evaluation of Software Documentation Formats", Journal of Systems and Software, Volume 9, No. 2, pp. 167-207, 1989.

[56] Moher, T.G., Mak, D.C., Blumenthal, B., and Levanthal, L.M., "Comparing the Comprehensibility of Textual and Graphical Programs", 5th Workshop on Empirical Studies of Programmers, pp.137-161, USA, 1993.