Syntactic Sugar Programming Languages' Constructs - Preliminary Study

Mustafa Al-Tamim[#1], Rashid Jayousi[#2]

Department of Computer Science, Al-Quds University, Jerusalem, Palestine

[1]maltamim@science.alquds.edu

00972-59-9293002

[2]rjayousi@science.alquds.edu

00972-52-7456731

Syntactic Sugar Programming Languages' Constructs - Preliminary Study

Mustafa Al-Tamim[#1], Rashid Jayousi[#2]

Department of Computer Science, Al-Quds University, Jerusalem, Palestine

[1]maltamim@science.alquds.edu

[2]rjayousi@science.alquds.edu

***Abstract*— Software application development is a daily task done by developers and code writer all over the world. Valuable portion of developers' time is spent in writing repetitive keywords, debugging code, trying to understand its semantic, and fixing syntax errors. These tasks become harder when no integrated development environment (IDE) is available or developers use remote access terminals like UNIX and simple text editors for code writing. Syntactic sugar constructs in programming languages are found to offer simple and easy syntax constructs to make developers life easier and smother. In this paper, we propose a new set of syntactic sugar constructs, and try to find if they really can help developers in eliminating syntax errors, make code more readable, more easier to write, and can help in debugging and semantic understanding.**

***Keywords*— Programming languages, syntax, constructs, syntactic sugars, syntax errors, ambiguity.**

I. INTRODUCTION

Developing and writing software application is common daily activity done by hundreds of thousands of developers and programmers as the demand on software applications is increasing to meet the technical revolution.

Enterprise software applications development using programming languages (PL) requires extensive code writing. Such applications have complex functionality and business logic for developers to focus on. Valuable portion of developers' time is spent writing repetitive keywords and determining code building blocks' scopes that can be ambiguous for them to follow up and debug, also it may generate many syntax errors that need extra efforts to find and fix. In addition, source code reading and semantic extraction by developers is not easy task when it's not their own code. Students who learn programming languages in universities and schools face similar issues in code ambiguity and syntax errors. These issues could cost programmers hours to fix syntax errors especially if they lack experience maturity to help them in code debugging and memorizing syntax keywords and complex structures. The problem becomes evidently visible when developers use remote access terminals like UNIX or simple text editors where no advanced IDE (Integrated Development Environments) and coding wizard available.

Syntactic sugar [4] enhancements on programming languages syntax constructs is one of the approaches used to enhance syntax and help in making it more readable, easier to write, less syntax errors and less ambiguous.

In this research, we proposed new set of syntactic sugar constructs that can be composed by a mixture of existing constructs obtained from some programming languages in addition to syntactic enhancements suggested by us. Through our work as software developer, team leaders, and guiding many students in their projects, we noticed that developers write a lot of repetitive keywords in specific parts of code like packages calling keywords, attributes access modifiers, code segments and building blocks' scopes determination symbols and others. One case example, the usage of curly braces "**{ }**" to determine program's building blocks (class, method, if, for…etc.) scopes in the same program, can make it difficult to distinguish the method scope from its internal control statements scopes especially in the case of missed opening or closing symbol.

This kind of repeated keywords and ambiguity can cause many syntax errors, and make it difficult to debug and understand the code semantic. This consumes portion of developers' efforts and time especially when using text mode development environment. This motivated us to search for syntactic sugar constructs that can help in enhancing programming language syntax in order to use less repetitive keywords, better scope determination symbols, better exception handling, and more readable code with less writing efforts to make developers work easier with more focus on business logic implementation. The questions we try to answer in this research: Is syntactic sugar constructs help in rapid development with less syntax errors? Can they make code more readable and easier to write? Do they help in semantic extraction?

Research results show positive indicators for using syntactic sugar in writing application source code. In the following section, we review the previous work done in this field. In section 3, we describe the methodology used to extract the syntactic sugars constructs set. The suggested syntactic sugar constructs are shown in section 4. Constructs validation case study is explained in section 5. In section 6, we show and discuss the case study results, and in section 7, we conclude and suggest future work.

II. BACKGROUND

The term "Syntactic Sugar" was found by the British computer scientist Peter J. Landin [4], this term describes making programming languages syntax user friendly and offer alternative syntactic expressions to language common constructs to be sweeter and written in simpler way without affecting the semantic [4] [5].

Syntactic sugars were used in many programming languages to offer new set of features in certain areas. In [4], *C# 3.0* was provided with new features to support LINQ as functional paradigm. These features were classified as syntactic sugars that help in cutting down the repetitive code tediousness. *W3C OWL Web Ontology Language* was extended by *OWL 2* [5] where OWL 2 added extra syntactic sugar to make common patterns and statements in OWL easier to write as the case of the disjoint union of classes.

The *OCL* language in [7] was extended by syntactic sugars as its concrete syntax is verbose and hard to be read, the authors added new syntactic sugars extracted from math and logic depending on positive results authors got from using the syntactic sugars within workshop notes and formalized due to UML 1.4.2 standard [15].

Java like languages (*Java, Scala and C#*), introduced many simple syntactic sugar that were used to reduce syntax complexity, as well as shortening and cleaning the code like omitting empty type parameters list in classes and methods, omitting empty arguments lists, and using special identifiers (_) for un-referenced parameters. [11] Describes *Liskell which* is a new syntax for Haskell that provides programmers with a set of syntax sugars to eases programming (Simple List, The Dispatcher Namespace, syntax sugar for defining macros "*defmacro*" and others).

Syntactic sugar also used within Aspect Oriented Programming in a language called *RE-AspectLua* which is a new version of *AspectLua* [13]. Authors used syntactic sugar to reduce the number of code lines needed to define aspect interface and associate it. In [9], syntactic sugar was used in Java based embedded domain specific languages (EDSL) to implement sugar methods to replace the Java noisy syntax and non domain related code used to create and set up domain specific objects. The XML document query language "*XQuery*" in [12], used syntactic sugar to offer shorter constructs for common and certain expressions (*The Empty Function, Quanti_ed Formula, FLWOR Expressions, Coercion)* to replace the complicated syntax used in research and education. *RhoStratego* language used syntactic sugars to code un-ambiguity by replacing parentheses with angle brackets [14]. It also provided syntactic sugar for concurrencies, lists, and tuples. In [8], *OpenC++* is C++ extension. OpenC++ provided syntax sugar for matrix manipulation library to define matrix as an array with initialized values which is not possible in regular C++, and a new kind of loop statement using "*forall*" notation to loop over all matrix entries in shorthand way. Authors of unfamiliar TEX language [6] provided syntactic sugars to make TEX constructs about the loop, the switch, array addressing, and keyword parameters closer to high level programming languages constructs like Pascal to be easier for users. All the related work described previously was focusing on enhancing certain syntax constructs partially to add support for specific concepts like supporting functional paradigm in object oriented, add shorthand methods and constructs, decrease code verbose and un-ambiguity) [10]. In our work, we tried to make enhancements using syntactic sugars on general level for the most common abstract constructs that are used in both object oriented and procedural programming languages paradigms. We propose using syntactic sugar to eliminate syntactic errors, reduction of keywords, better semantic extraction, code debugging, and make remote development easier.

III. CONSTRUCTS SELECTION

Constructs selection methodology that we used to select and enhance the syntactic sugar constructs set depends on two factors: Usability frequency of constructs in writing programs, and Object Oriented Programming (OOP) Relevance. We determined the common and widely used abstract programming constructs [1] that are classified under these factors.

The abstracted common constructs we used and related to usability frequency are: *Method (function) definition, Looping construct, Selection construct, Building blocks scopes determination, and Exception handling variables scope constructs*.

In the abstracted common constructs relevant to OOP, we focused on constructs that represent the main OOP concepts like inheritance, encapsulation, polymorphism, and relations. The constructs are: *Class, Inheritance, Using Methods as Constructor, libraries and packages usage, Class Attributes access modifiers, Methods access modifiers, Objects Collection Iteration, Object instantiation, and Object / Method messages passing (calling) format.*

Using the abstract constructs set; we extracted the actual syntactic constructs from set of programming languages. We selected 5 programming languages and extracted the actual syntactic constructs from them. The programming languages were selected upon: 1) Languages usage and spreading. 2) Languages families and development. The Selected programming languages are: Eiffel, Python, Java, C#, and Ruby. We considered selecting languages that are developed on top of others older languages or their syntax is a mix of other pre-exist languages. This to make balance between old and new programming languages, and to cover syntactic constructs that are used in large set of programming languages [2]. The final set of the syntactic sugar constructs proposed was a mix of syntactic constructs we extracted from programming languages which considered to be widely used, in addition to a set of syntactic enhancements suggest by authors as syntactic sugar constructs.

The first constructs set was general and included many alternatives for the same abstract construct. To narrow the selection and form the new syntax constructs set, we followed questionnaire approach to get people who use programming languages (programmers, developers, students…etc.) opinion

and know their recommendation of which constructs are better upon their experience and expectations.

We distributed the questionnaire over programming professionals and students in Palestinian universities and companies in west bank – Palestine. The populations and sample size was calculated depending on a report of ICT working forces in Palestine [3]. The sample size was: ICT Professionals: 77, ICT Students: 93. Number of distributed copies is 600, collected were 251 as follows: ICT Professionals: 79, ICT Students: 172.

The analysis of the results obtained from the collected questionnaire showed that 14 out of 15 questions' answers were Java constructs selection, only 1 constructs was from another languages (Ruby). The results helped us in realizing a fact the people usually prefer what they know and resist change (change management); they answered in a way that didn't nominate new easier constructs set, we concluded this using "*Percentage Distribution of ICT Professionals According to Technical Skills*" statistics in ICT working forces in Palestine report [3].

Results directed us to modify our methodology by nominating a set of syntactic sugar constructs from the extracted and enhanced set we assume that it help in improving code syntax and achieve all objective we try to approve (the nominated constructs set is explained in section 4). Then we asked people to practice them, and received their feedback as explained in section 5.

IV. Syntactic Sugar Constructs Set

Upon modification done in the methodology, we nominated set of syntactic sugar constructs to produce new partial syntax for programming languages common constructs. The constructs selection criteria were: 1) Reduce repetitive keywords. 2) Make construct shorter to write, Close to natural language and standard like UML notation. 3) Offer many writing form alternatives for the same construct. 4) Enhance constructs scope symbols to make code more readable and less ambiguous.

The following table summarizes all selected and enhanced constructs:

TABLE I

Selected and Enhanced Syntactic Sugar Constructs

Enhanced Constructs	Suggested Syntax	Comments
Class Inheritance Construct	**class** ChildClass **->** ParentClass // UML notation **class** ChildClass**:**ParentClass	Offers code reusability, shorthand, and maintenance.
Class Instantiation Construct	**myInstance** = **MyClass**();	Keyword reduction
Method Definition Construct	**def** methodName(int size, Object obj) int x = 5 + size; **return x;** **endef**	Used simple construct to define a method where the return type is not needed.
Method Calling Construct	**instanceName**.**methodName;** // calling method without parenthesis **instanceName**.**methodName();** // calling method with parenthesis **instanceName**.**methodName2(5, objInst);** //calling method with parameters and parenthesis **instanceName**.**methodName2 5, objInst;** //calling method with parameters and without parenthesis	Many alternatives to call a method from class instance and message passing
Method Execution on Class Construction Construct	**class** MyClass **create executeMeMethod{** **def executeMeMethod**() system.out.println("I'm executed on instance"); **endef** **}**	Used to execute a method on class instantiation without using constructors or if no constructors / defaults constructor is available.

Looping Construct	**5:times do ref // loop 5 times**
	System.out.println("Val: "+**ref**+" in: "+arr[**ref**-1]);
	end
		Used to Loop a block of statements or array entries number of times in simple way. "ref" is optional.

Object Collection Iteration Construct	**myCollection:each do ref // iterate myCollection**
	System.out.println("Hi, I'm looping…"+**ref**);
	endEach
		It iterates over collection of objects or any type derived from collection type in easy way

Selection Construct	**choose(a){** **case 1:** System.out.println("One…"); }	The keyword used to me more close to human natural language

Packages / Modules Calling Construct	**import: java.io.*; // write import only once for all**
	java.util.*;
	java.lang.*;
		This to reduce repetitive "import" keywords

Variables Access Modifier	**class** ClassName**{** **private:** // private attributes int a = 1; String b; **public**: // public attributes File file = new File(); double length; **// the same for other access modifier** **}**	An enhancement to define many attributes with the same access modifier. *Close to C++*.
Method's Access Modifier	**def __privateMeth()**//2 underscores : private **def _protectedMeth()** //1 underscores: protected **def publicMeth()** //no underscores: public method	The access modifiers for methods are specified in simple way by using underscore(s) "_" at the beginning of method name.

Exception Handling Variables Scope	**try**{
	int nm = Integer.parseInt(br1.readLine());
	}
	catch(Exception e){
	System.out.println("num="+**nm**);//nm is accessible
	}
	System.out.println("num="+**nm**);//num is accessible
		We modified the scope (accessibility) of variables defined within the exception try block to be accessible outside the try block

V. The Exploratory Case Study

Measuring constructs efficiency is done by asking users to practice them. To verify the assumption of the suggested syntactic constructs set obtained upon modified methodology, we designed and executed exploratory case study. We conducted an exploratory case study with small sample size as we considered this experiment as an indicator to know if our assumption regarding the new constructs set was valid. We do not claim that results in this research are final, they are indicators. We were unable to make the experiment with large set of users because of many difficulties we faced: students were not interested to participate, their times and availability was hard, professional developers don't prefer to spend time in doing work outside their paid time and their availability is hard to be managed. The case study designed into two tracks: The first track was with computer science students. We introduced the new constructs to them with simple training, get their feedback through an interview, then we asked them write some programs using the new constructs set designed upon their university courses with different difficulty levels. In this track we wanted to measure the percentage of syntactic errors and difficulty in writing programs by students, and if the new constructs set can help in eliminating errors and code writing efforts. The other track was with programming professionals who work in software industry. We gave them set of shuffled programs (*some of them written using the new constructs and the others in old Java syntax*), and asked them to debug and extract their semantic. Then we introduced the new constructs set to them, and did an interview to get their feedback. In this track we wanted to verify if the new constructs can help in semantic extraction, debugging, and making the code more readable. We implemented the new constructs set as syntactic extension on top of Java 1.5 syntax using parser generator tool called JavaCC [16] in addition to a very simple integrated development environment (IDE) used by participants to write programs using the new syntax constructs set. After completing the case study execution, all data were collected (interview answers, written programs, errors' log files) and analyzed to get results as explained in section 6.

TABLE II

Interview Questions' Answers Summary

Questions	Students' Results	Professionals' Results
1- Do you believe that using the new constructs will save efforts in writing code especially in case of repetitive keywords (import, access modifiers…etc.) and shorter looping constructs?	Agree	Agree
2- Do you think that using new constructs will help in decreasing syntax errors as result from saving repetitive keywords and distinguish scope using different identifiers?	Totally Agree	Agree
3- Do you agree that using new constructs will make the code debugging easier?	Agree	Totally Agree
4- Do you think that the code will be more readable using the new constructs?	Totally Agree	Totally Agree
5- Are the new constructs can help in extracting the program semantic from just reading it with minimal execution efforts and without the need for executing it many times and debug it to understand its functionality?	Agree with Reservation	Agree
6- Is it true that the new construct can help in producing programs with less number of code lines (shorter syntax)?	Totally Agree	Totally Agree

VI. Case Study Results

Analyzing the data collected in all case study tracks showed encouraging results and positive indicators that support research assumptions. Interview summarized answers in Table 2 for both students and professionals showed that the new syntax constructs set affect on decreasing the syntactic errors and making the code more readable and shorter.

They help in saving code writing efforts, using less repetitive keyword, and make code debugging easier, and extract programs semantic in easier and faster way. In students' case study track (*Programs Writing*), and depending on the log data generated by the parser, we extracted and counted errors happened in each program for each student. We classified the errors to two types: errors occurred in old Java syntax, and the errors occurred in the new suggested constructs. This classification is to measure the percentage of errors occurred by each type and to check if there was any improvement. Prior to analyzing the results in Fig. 1, we counted the number of each constructs type used in each program and summarized them. This is to check if the ratio of generated errors from each constructs type is reasonable to the number of constructs used in each program. As shown in Fig. 2, we notice that the used new constructs form 39% of whole BubbleSort program constructs and old construct are 61%, but if we looked at percentage of errors in Fig. 1, we find that new constructs caused 8% of total errors in this program while the old constructs caused 92%. The same observation can be noticed for the other programs. Also, It is noticed that whenever the program becomes longer (number of syntax lines is higher), new constructs are used more in the syntax as shown in Fig. 2. This show another observation: the new constructs are used in higher percentage in longer programs while they generate fewer errors, this mean the total errors count in the programs will decrease due to new constructs usage.

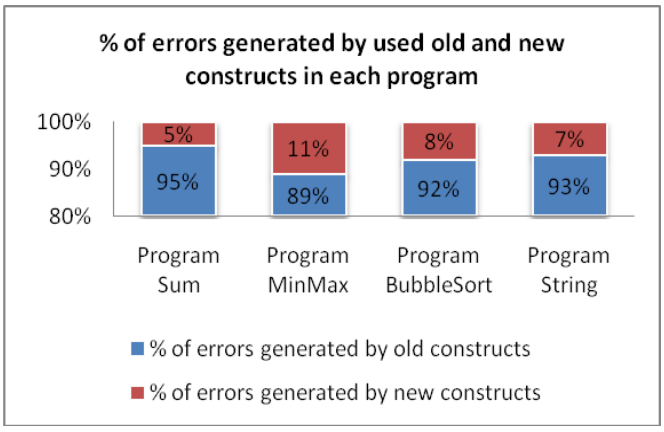

Fig. 1 Percentage of generated errors

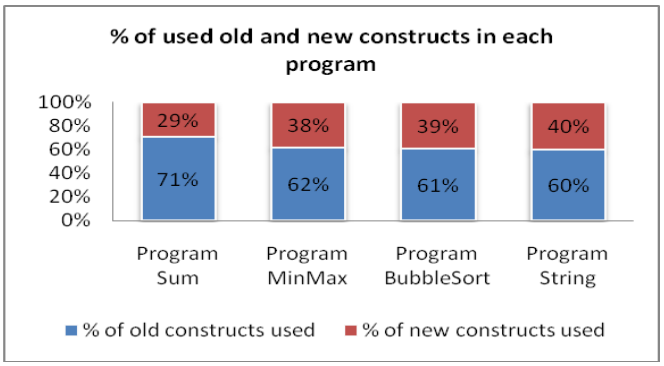

Fig. 2 Percentage of constructs used in each program

In professionals track (Semantic Extraction), answers were collected and graded in scale start from 1 to 3. 1 means the extracted semantic is far from the correct answer, 3 means the semantic is correct and accurate. We calculated average answer grade for each program for all answers using the following equation:

$$A_{avg.} = (\sum A(1...n))/n$$

Where A: The professional answer grade and n: number of participant professionals

Then, we calculated "***Accuracy Ratio***" to show how many the $A_{avg.}$ for each program is close to the complete accurate answer grade which is 3. The Equation used:

Accuracy Ratio $= A_{avg.} / 3$.

From results in Table 3, we concluded that the new constructs helped in extracting more accurate programs semantic than the old constructs. The new constructs results show the lowest accuracy ration was 83%, and the highest was 100% with two programs had accuracy of 93%. In the old constructs' programs results, the lowest accuracy was 53% which is much less than the lowest new constructs accuracy result, and the highest was 93% and not 100% as the new construct highest accuracy. It is important to note that professionals asked to extract the semantic of programs without any previous knowledge about the new constructs while they had enough knowledge about the old constructs as all participants were Java developers.

TABLE III

Semantic Extraction Results

Programs	Constructs Type	A avg.	Accuracy Ratio
P8	Old Syntax	1.6	53%
P9	Old Syntax	2.5	83%
P3	Old Syntax	2.5	83%
P1	Old Syntax	2.6	87%
P5	Old Syntax	2.8	93%
P10	New Syntax	2.5	83%
P7	New Syntax	2.6	87%
P6	New Syntax	2.8	93%
P2	New Syntax	2.8	93%
P5	New Syntax	3	100%

From the results, using different scope determination symbols for each construct help in reducing errors and make code more readable and less ambiguous. And the effect of new syntactic sugar constructs on semantic extraction was higher from professionals' perspective.

VII. Conclusion and Future Work

In this work, the new constructs set with syntactic sugar showed positive indicators that can help in producing less syntactic errors and repetitive keywords, more readable, shorter, easier to write, debug, and clearer code, in addition to better scope determination, and more accurate semantic understanding. We recommend considering these results in the design of new programming languages' syntax.

In future work, we need to extend the constructs set to include new constructs and increase experiment sample size to be larger and longer period. One way we intend on doing is to teach the constructs during a university course for several semesters to get more representative evaluation.

References

[1] Peter D. Mosses, "*A Constructive Approach to Language Definition*", Journal of Universal Computer Science, vol. 11, no. 7, 2005, pp. 1117-1134.

[2] Éric Lévénez, *Computer Languages History*, At http://www.levenez.com/lang/lang_a4.pdf, 2009.

[3] The Palestinian IT Association of Companies (PITA), *Assessment of the Palestinian ICT Workforce*, At http://www.pita.ps/newweb/pdfs/local_2008.pdf, 2008.

[4] Ahmad Emad Mageed, "*The Evolution of the C# Language: The Impact of Syntactic Sugar and Language Integrated Query on Performance*", A thesis submitted to the Graduate Faculty of Auburn University, Auburn, Alabama, 2010.

[5] Christine Golbreich and Evan K. Wallace, *OWL 2 Web Ontology Language: New Features and Rationale*, W3C Working Draft 02, 2008.

[6] Kees van der Laan, "Syntactic Sugar", *Dutch TEX Users Group (NTG)*, AJ Schagen,The Netherlands, 1992.

[7] Jörn Guy Süß. "Sugar for OCL". *Proceedings of the 6th OCL Workshop at the UML/- MoDELS Conference*, 2006. Pp. 240–251.

[8] Shigeru Chiba, "*OpenC++ Programmer's Guide for Version 2*," Xerox PARC Technical Report, 1996

[9] Steve Freeman and Nat Pryce, "Evolving an Embedded Domain-Specific Language in Java", *OOPSLA '06 Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, 2006

[10] Philippe Altherr and Vincent Cremet, "*Abstract Type Constructors for Java-like Languages*", At http://citeseerx.ist.psu.edu/viewdoc/similar?doi=10.1.1.90.6424&type=ab, 2006.

[11] Clemens Fruhwirth, "*Liskell Haskell Semantics with Lisp Syntax*", At http://clemens.endorphin.org/ILC07-Liskell-draft.pdf, 2007.

[12] Jan Hidders, Philippe Michiels, Jan Paredaens, and Roel Vercammen, "LiXQuery: A Formal Foundation for XQuery Research", *SIGMOD Record*, Vol. 34, No. 4, 2005.

[13] Thaís Batista and Maurício Vieira, "RE-AspectLua - Achieving Reuse in AspectLua", *Journal of Universal Computer Science*, Vol. 13, No. 6, pp 786-805, 2007.

[14] Eelco Dolstra, "*First Class Rules and Generic Traversals for Program Transformation Languages*", Utrecht University, 2001.

[15] Object Management Group (OMG). *Unified Modeling Language Specification*, Version 1.4.2, Jan 2005.http://www.omg.org/cgi-bin/doc?formal/05-04-01.pdf.

[16] JavaCC, At https://javacc.dev.java.net, 2010