Text to Code: Pseudo Code Generation

Altaf U. Din[1(✉)] and Awais Adnan[2]

[1] AWKUM, Mardan, Pakistan
altafkhattak@gmail.com
[2] IMSciences, Peshawar, Pakistan
awaisadnan@gmail.com

Abstract. The evolutions in programming from machine language to these days programming software have made it easy, to some extent, to develop software but it is not as easy as programming in natural language. In order to transfer natural language text to any programming language code, it felt necessary to first transform natural language text into pseudo code algorithm then with the help of right API library, such algorithms can be transform into any programming language code. The main aim of this research work is to produce pseudo code from text however this work is very loosely bound to natural language processing. Main components of this proposed work is text analyser that utilizes language tools (spelling check, grammar check) to remove type errors and then eliminate different ambiguities. For this step of ambiguity removal, an adaptive solution is proposed that learning from manual assistance. Once the text is cleared, pattern matching techniques is applied to it and later on parsed into a pseudo code. The concept model is tested with user scenario approach and also practically implemented by developing a prototype. This model is examined using 100 examples of different categories and achieved 73.

Keywords: Pseudo code generation · NL to programming code · Intelligent system · Code extraction from text

1 Introduction

How fascinating it would be if a person, who is not a programmer or coder, in need of certain specific application or program and all he does is to tell the computer about his requirements. The computer then creates the program according to his needs. The question that arises that is it possible to build such intelligent system? In couples of decades ago the answers were either no or quite vague for generating programming language code from natural language. However these days, with the advancement of technology, there is possibility that in near future people might come across such system that will facilitate them to produce software by using their natural language. The programming techniques are very tough for general users and, in today technology most of users of computer and smartphone are aware of customized software. It requires a lot

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2019
Published by Springer Nature Switzerland AG 2019. All Rights Reserved
P. C. Vinh and A. Rakib (Eds.): ICCASA 2019/ICTCC 2019, LNICST 298, pp. 20–37, 2019.
https://doi.org/10.1007/978-3-030-34365-1_3

of tools and effort to write a program, but it needs a user to understand the programming concepts. Moreover the programmers must be able to know more than one programming language styles where their libraries, structure and syntax might vary. Since the beginning of computing era, it is always in focus of researchers to make programming easier. Assembly language was introduced to use mnemonics and other prompts in order to get over with machine language programming. But still it was not enough, high level languages were introduced that could adjust most of the programming parts with words from English language. The High Level languages did provide ease and comfy to programmers, not only just making their code writing easy but it was also easy to learn as well. However the issue that still exist is that a novice programmer still has a long route to polish his skills in programming. An expert coder must have to keep himself up to date with the advancement in technologies as well as updates in programming languages. Moreover a layman is still paying huge sum of amount in order to develop certain software or program that somehow reflects upon his requirements. On the other hand the implementation style of different available programming language is also different from each other. On the other hand the implementation style of different available programming language is also different from each other. [1] There is need to provide users with a conventional way where they can state their requirement in natural language (i.e. English language) and the system is able to process the text and provide output is some form of pseudo code. Pseudo code can be transformed into any programming language code with the help of appropriate libraries or API. From the current advancement in other fields of computer science and enhanced techniques (for instance machine learning techniques, data and text mining techniques, natural language techniques etc.) it is assumed that generating code from natural language is possible now with so much development [2]. In this research work a conceptual modal has been proposed that fulfills the purpose of collecting requirement text, in natural language, from user and aims to provide a pseudo code that could easily be transformed to any programming language. The model is composed of four layers that are "*Application Layer*", "*Text Formalization Layer*", "*Translation Layer*" and "*Code Generation Layer*". The application layer is the interface where user will input text as requirement of a particular program. The text formalization layer is where the text is refined down for easily transferred to pseudo code. The translation layer is where pseudo code will be generated and code generation layer is where programming language code from pseudo will be generated. This research work is a small contributory step into long term vision of programming in natural language. The following is exploring the model in detail and implementing the model in order to observe the result.

2 Literature

The concept of bringing down the technology, to the level, where a layman can easily communicate with computer systems is not new. Since 1960 researches are working to create more natural language adaptable systems and applications.

SHRDLU [3] is considered as one of the earliest interface that could answer the questions of a user by manipulating the objects known as blocksworld. However that was an application to step into understanding the natural language. The intended work is more inclined toward programming with natural language. People have proposed their work time to time that requires input in natural language and transform into something that can be meaningful for programming or software development for instance Deeptimahanti and Sanyal [4] have proposed a semi-automatic technique in their work that helps developers to generate UML machine from normalize natural language requirements. Abirami et al. [5] have conducted their study on a classifier that is supposed to separate Functional Requirements (FR) from Non Functional Requirements (NFR), from text that is written in natural language. Siasar djahantighi et al. [6] has worked with same the theme of exploring natural language to produce SQL queries. Their technique is mainly based on a parse tree that is able to identify verb, noun and entities in a statement and generate SQL query. Moreover it is worthy to mention the Natural Language Interface to a Database (NLIDB) [7]. That is among one of the earliest concept mentioned for the ease of general mass to use database application. As the user uses an interface with which they would communicate in their own language with the database application. There is very little related or similar work available where code is generated from plain text in natural language. Price et al. [8] introduced naturaljava. The purpose of naturaljava is to provide such interface for users where they are able to type a command in natural language without any worry about the syntax and the output will be produced in java code. The architecture of naturaljava is shown in Fig. 1. NaturalJava is composed of three main components that are sundance, PRISM and TreeFace. Sundance grabs the input in natural language and generate case frames. PRISM is the case frame interpreter that gathers operations from case frames and Treeface is Abstract Syntax Tree (AST) manager. PRISM uses Treeface to manage the syntax tree of currently java program in progress. Vadas and Curran [9] have mentioned few problems related to naturaljava in their work. According to them Naturaljava can process the accomplishment of

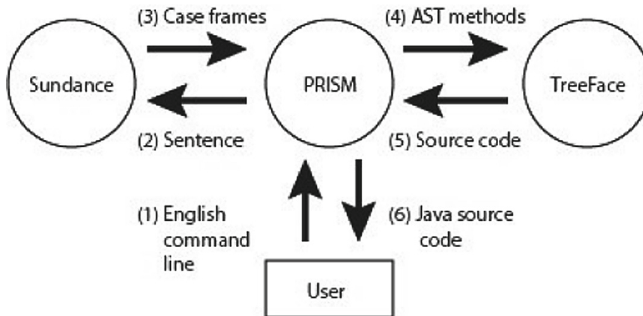

Fig. 1. Architecture of NaturalJava

one sentence at a time and it can only produce code in java language and cannot be adopted to any other language. Their proposed system architecture is given below in Fig. 2. The input text will be parsed by Combinatory Categorical Grammar (CCG) parser [10] and the output of CCG parser will be transformed into more generic representation of text. Afterward a python code output will be generated. There is a dialog session with user in every step as to handle ambiguity but there is no mechanism that will record the dialog session for future reference so that the system should not repeat the same dialog session for any similar scenario. Nadkarni et al. [11] have introduced a conceptual model that has approach of writing an algorithm in natural language and it will be converted to c language code. The input should not be in absolute natural language as they tend to force semi natural language algorithms to be converted into c language code. The challenges for code generation from algorithm typed in semi natural language are the polysemy of algorithms is hard to process, different structure of different programming languages and flexibility. After the user provides input through user module then next stage is to apply standard or basic language processing steps in which each line is explored as well as each word of each line. Then the interpreter section is divided into two steps the first step is to spot the

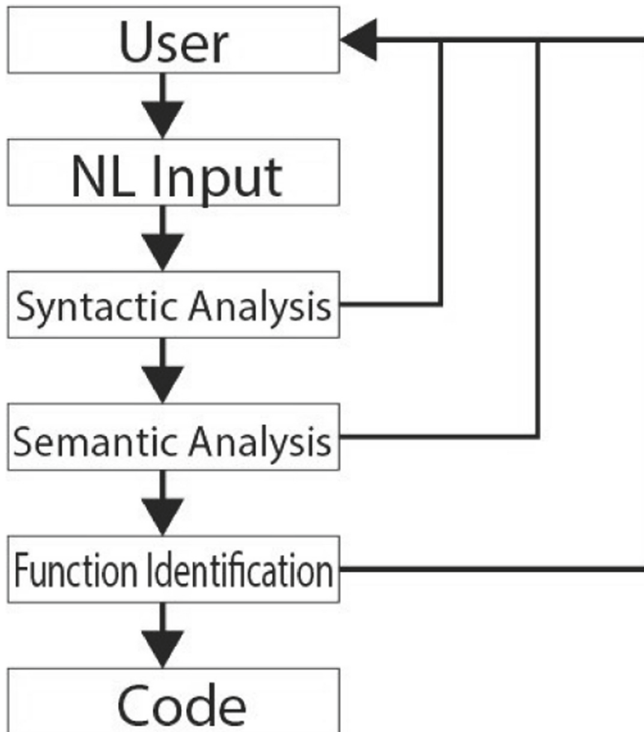

Fig. 2. System architecture

keywords while in next step, if it is identified properly, then it will transformed into code. Meanwhile the synonyms of the words will also be checked and on top of that it will also facilitate users to mould the system according to their algorithm writing style. They did their implementation through user scenario example and in the end they concluded with showing how flexible their modal is but the issue is that input should be provided that is projecting syntax in natural language style hence the input mechanism is not flexible toward as ease of using natural language. Thomas et al. [12] proposed a model of a compiler that will convert natural language into source code of any programming language. They have used the pipeline architecture that will utilize three different databases in order to produce output. Their compiler works in three phases and each phase will overcome the following three issues. proposed model showing natural language compiler interaction. The first phase is named as "Find Meaning" where an array will be created of the text and words will be checked against their available commands and variables. The next phase name is "solve commands" that will check for commands that may or may not need input or output. The last phase is labelled as "Convert intermediate code" that is basically parsing according to each pattern matched in above phases.

- Variations of word
- Context of word
- Generation code in any language

According to their work the NLP compiler has pipeline architecture hence it can be modified a little and transform into a compiler that will convert natural lan-

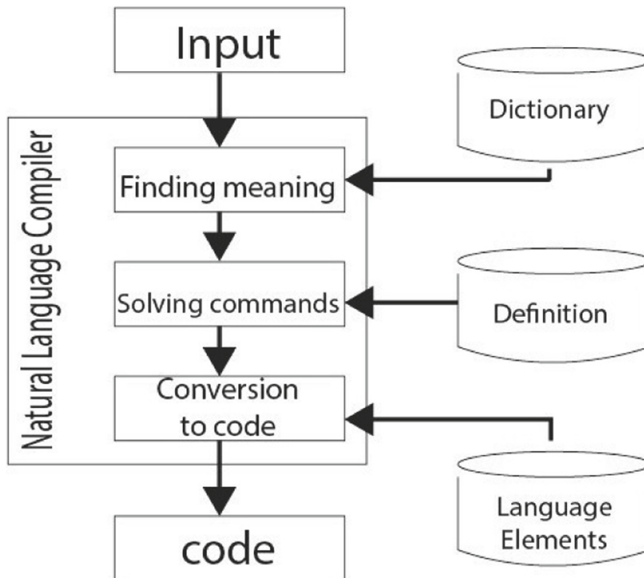

Fig. 3. NL compiler interactions

guage into programming code. The Fig. 3 is the After going through the study of previous work, it has been found that either the input method is not entirely into natural language or the output generated is in a specific programming language. The proposed method will tackle with the mentioned limitations of previous work by accepting input as in pure natural language and output should be in some form of vague code that could be translated to any other language easily. On the other hand there is no mechanism that should explicitly handle the ambiguity. Anjali and Babu Anto [13] argues that solution for ambiguities is very complex task however different methods of NLP (such as machine learning techniques, machine translation or methods of information retrieval) may provide a good resolution in future.

3 Proposed Model

The conceptual model for generating code from plain text, written in natural language, is composed of four different layers.

- Application Layer
- Text Formalization Layer
- Translation Layer
- Code Generation Layer

Figure 4 shows the model of proposed solution in next sub section.

3.1 Application Layer

The application layer is actually the interface for the user. The UI should be user friendly. Moreover the view of interface should have capacity for the following basic points. **"Input Space":** An input area for the user to type in their program requirements in natural language. "**Trigger**": Once a user is done with the input, a trigger (a button for example) should be provided to grab user text from input space and send it to processing. "**Result display space**": After processing text to mock code there should be space provided to display the pseudo code. "**Assistant wizard**": If the system is unable to generate pseudo code then it should initiate Wizard for the user. The wizard will guide the user through dialog session in order to figure out any issue. "**Code generation trigger**": After successfully generating pseudo code a trigger should be provided for users to generate any specific programming language application. A trigger here can be a dropdown list of different programming languages options (i.e. c++, java, php etc.)

3.2 Text Formalization Layer

From text formalization it means to bring down the informal/usual form of text into a standardized form. The reason for putting text into a proper format is that it will make way easy for pseudo code generation. **"Spelling and grammatical**

mistakes removal" is quite easy these days with the advanced technique however **"ambiguity removal"** is still a challenge. There are some mechanisms available to remove certain types of ambiguities but these techniques are not enough and more work requires as it is complex area of field [14]. In this model ambiguity removal have been placed explicitly as with the passage of time new techniques can be added to the system in order to make it more robust. The process of ambiguity removal is shown in Fig. 5. "**Text refining**": The purpose of text refining is to put the sentences in order as well as replace the "typical words" with associated "programming word". The term typical words are used here to expresses bunch of words regarding any keyword that is usually used in programming. For example Table 1 shows keywords and the typical words regarding keyword. The process of text refining stage is highlighted with the help of Fig. 6.

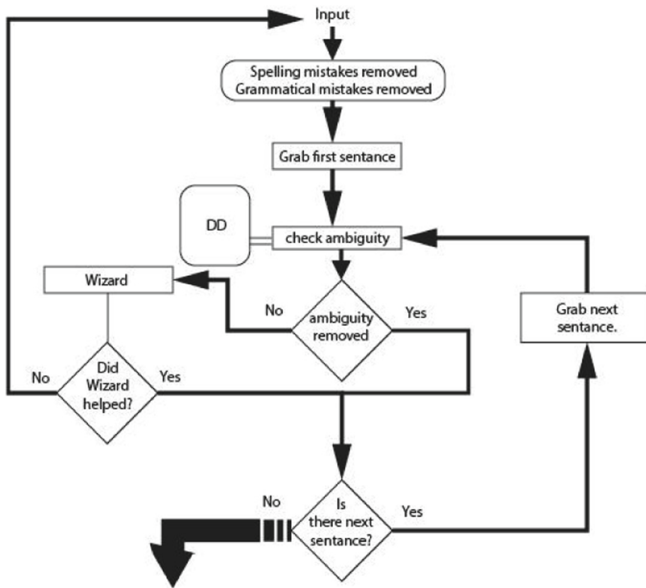

Fig. 4. Ambiguity removal

Table 1. Programming word and its matches

Programming word	Typical words
if	Condition, whether ... etc.
else	Otherwise, if not, ... etc.
loop	Iterate, repeat, recur ... etc

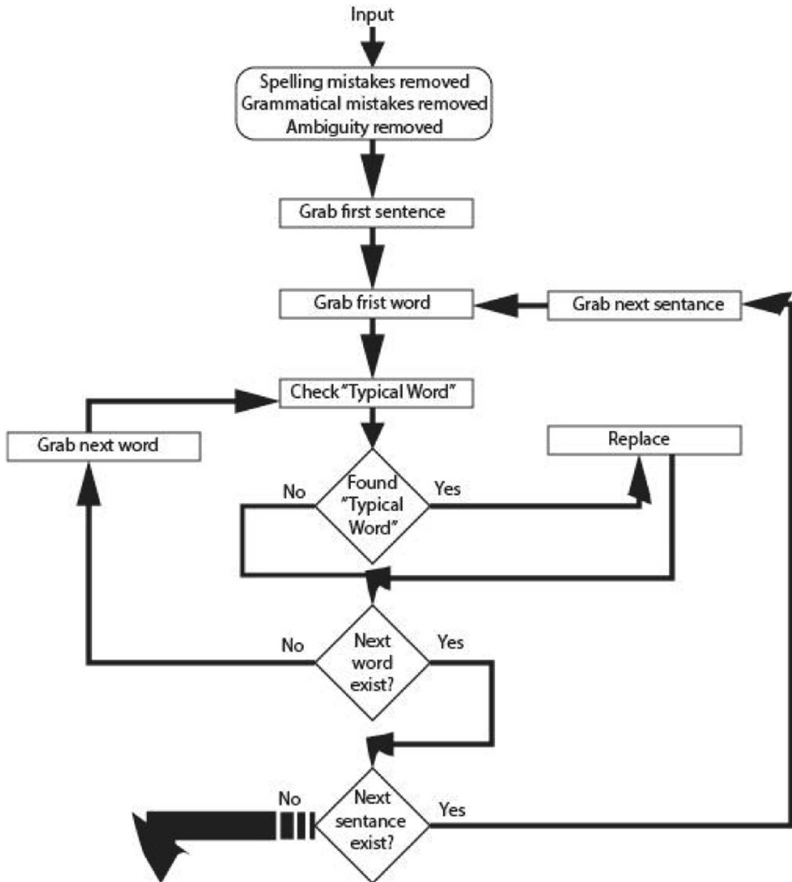

Fig. 5. Text refining

3.3 Translation Layer

The translation layer is divided in two different phases that are pseudo code generation phase 1 and phase 2. In "Pseudo code generation phase 1" the "word patterns" will be matched and replaced. Word pattern is actually occurrence of certain words in such an order that will comply with syntax of a programming style. Considering the example of if-else statement. The syntax of if-else statement is

If(condition)
-do something-
Else
-do something else-

Now processing the statement "if answer is 1 then print Earth else print rest of the Universe." The pattern here is, anything between word "if" and "then" is usually condition. Anything between word "then" and "else" has to happen when condition is true. Anything between "else" and full stop has to happen if condition does not meet as expected. So looking at the word patterns in the above statement, the pseudo code it will generate as

if(answer is 1) **Print** Earth **else Print** rest of the Universe

Figure 7 shows the process of phase 1.

In the "**pseudo code generation phase 2**" the generated pseudo code so far will be checked if each line is in arrangement of a programming code style. If there is any line that does not confirm the style then wizard will be invoked to get the issue out of that line. Figure 8 shows the process of phase 2.

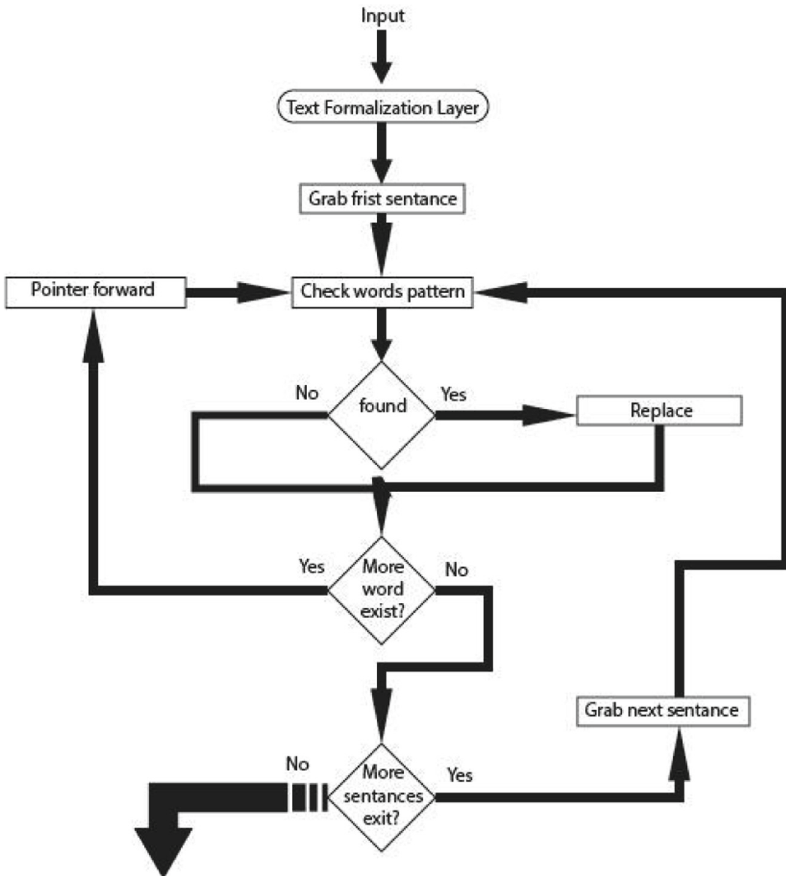

Fig. 6. Pseudo code generation phase 1

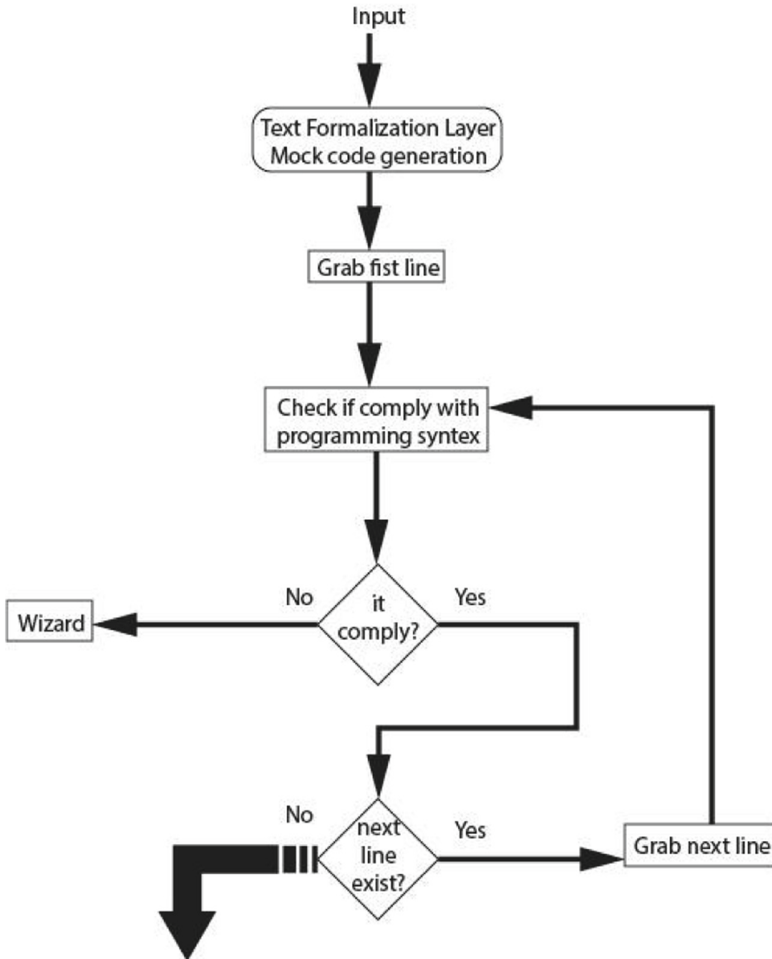

Fig. 7. Pseudo code generation phase 2

3.4 Code Generation Layer

The code generation is subjected to be completed in future as the main aim of this work is to deliver pseudo code from natural language input. Because if a system is able to convert requirement provided in natural language into algorithm then, with the aid of right API and library, it will not be difficult to generate any programming code from algorithm. Apart from the layers there are three main components to the system that are data dictionary, variable check points and assistant wizard. **Data Dictionary** The data dictionary is actually the database of the system. **Variable checkpoints** Variable checkpoints are the way to handle variable declaration. First check point for variable declara-

tion will be after text refining stage. At this point all the noun words will be declared as variable. The next variable checkpoint is after pseudo code generation phase 2. At the stage the system will discard all the noun variables that are not used. Meanwhile it will declare some more variables that are in use but not declared. **Assistant Wizard** Assistant Wizard is a dialogue session with the user. Assistant Wizard will be initiated when the system is unable to process the text provided. It will guide the user as well as collect information from the user and with the successful completion of session it will record the scenario for future reference.

4 Implementation

A user scenario approach of implementation should be carry out before practical implementation in order to understand the mechanism thoroughly.

4.1 User Scenario Approach

The following text should be processed out through each layer of the model manually. *"A user is enter marks. Check for marks if it is great than 49 then disply pass otherwise it should disply fail."*

The spelling and grammatical mistakes are deliberately left there for demonstration purpose. The first layer is "application layer". A user will be able to provide the text through application layer. Next is the text formalization layer. The first section of text formalization is to remove the spelling mistakes. The underlined words have spelling mistakes in the input text. "A user is enter marks. Check for marks if it is great than 49 then disply pass otherwise it should disply fail." After removing spelling mistakes the statement will be converted to: "A user is enter marks. Check for marks if it is great than 49 then display pass otherwise it should display fail." Then grammatical mistakes will be removed as the underlined parts in the statement have grammatical mistakes. "A user is enter marks. Check for marks if it is great than 49 then display pass otherwise it should display fail." After removing grammatical mistakes the statement will be converted to "A user will enter marks. Check for marks if it is greater than 49 then display pass otherwise it should display fail." After getting rid of grammatical and spelling mistakes, the ambiguity will be checked. Looking at the statement the ambiguity lies where it says "it should display fail." The system will not be able to decide where "it" directs to. So removing the ambiguity the statement will take the face as:
"A user will enter marks. Check if marks is greater than 49 then display pass otherwise display fail." The next stage is "Text Refining" where the typical words is spotted and replaced. The underlined words in the statement includes typical words. "A user will enter marks. Check if marks is greater than 49 then display pass otherwise display fail." The typical words can be replaced as enter to input, is greater than to $>$, display to print, if remains as if and otherwise to else. The text will be converted to: "A user will input marks. Check if marks

$>$ 49 then print pass else print fail" The variable checkpoint will declare words user and marks as variable as they are noun and proceed to translation layer. In the first phase of pseudo code generation the system will check for the words pattern and replace it with the vague code accordingly. For instance in the above statement the word phrase of first sentence that says "A user will input marks." The word input indicated there should be variable around where value should be stored. At the left side of word input is "will" which is a modal verb that is kind of base for the verb and not noun while on the right side "marks" is noun and there is no preposition etc. between the word input and marks hence marks should be a variable. As the word "input" indicates someone will enter the input through keyboard because if there was word phrase "stores", "have value", "keeping number" and so on that would meant to store some value in marks variable. So the pseudo code for it will be "input marks= from keyboard". The full stop indicates to jump to new line. And the word phrase in next sentence is "if marks $>$ 49 then" the word "if" indicates there is conditional statement of if-else. Usually anything between if and then is considered as condition and should be put in small brackets e.g. if(marks $>$ 49). Anything between then and else is body of the if statement section when condition is true and it should be kept between and . So getting the phrase of then to else and put it in order it will be displayed as print pass. There is else word included then the word phrase is anything between else and full stop is body of else. This in this example is print fail. Putting it all together, after adding start and end to the algorithm generated, the output of the pseudo code generation phase 1 will be

```
Start
Input marks=from keyboard
if(marks > 49)
{
Print pass
}
else
{
Print fail
}
End
```

The next phase of pseudo code generation it will be checked if it complies with programming code style.

After successfully passing through phase 2 the variable checkpoint will discard "user" variable as it is not used (Table 2).

4.2 Practical Implementation

For practically implementing the proposed model, an initial prototype application was developed using mySql and netBeans IDE. Around 20 people were asked to provide at least 10 different statements regarding four simple mathematical functions (addition, subtraction, multiplication and division) between two values. About 200 statements were collected and placed in their respective group of

Table 2. Generated pseudo code to programming code

Generated pseudo code	Programming code style
Start	Start of the program
Input marks=from keyboard	Input by user
if(marks > 49)	if(condition)
{ Print pass }	{ Do something }
Else	Else
{ Print fail	{ Do something else }
}	}
End	End of the program

addition, subtraction, multiplication and division. Most of the statements were almost similar but there were few statements that actually helped in building up small data dictionary of typical words regarding the four basic mathematical operations. A simple statement of "deduct var2 from var1 and show result to user" was processed through it and the resulting algorithm was successfully generated by the prototype application. Figure 9 shows the resulting pseudo code generated for the above mentioned statement. Likewise "twinkle twinkle little stars." was provided as input and Fig. 10 shows the result of invoking assistant wizard. Later on the database was updated more in order to process natural language text including conditional statements of if-else and "if student1 is greater than student2 then display student1 has gotten high marks." was provided as input to the prototype. Figure 11 shows it successfully processed the text.

5 Analysis

The system was able to produce output for different natural language text input against different programming statements. Although it did call upon wizard in order to resolve some issues that is due to the lack of mature data dictionary at this stage as well as no proper ambiguity resolution mechanism attached to the system, as the prototype is in initial phases.

100 different basic statements are tested on the proposed system out of which 73% are understood correctly and for the rest of 27% wizard was prompted. Details about the statement and corresponding result are illustrated in the Table 3 below.

Following are a few results observed from the above cases. Words like display, Print, Show, Get, Input, Output, subtract are identified with 100 Some words, which initially were considered simple, showed high level of error. One such word is times, which confuses multiplication and loop. Another such example is dividing. Initially it was considered simple words but experimental results shoes that system was confused between multiplication (e.g. divide 10 apples among 5 student 10/5 identified correctly, divide 2 cakes in to 10 pieces each, identified as

Fig. 8. Result for subtraction algorithm

Table 3. Results with and without wizard prompts

Type of statements	Statements tested	Result without wizard	Wizard prompted
Input statements	10	10	0
Output statements	10	9	1
Addition	10	9	1
Subtraction	10	9	1
Multiplication	10	7	3
Division	10	8	2
If statements	10	8	2
If-else	10	6	4
For loop	5	3	2
While loop	5	2	3
Do-while loop	5	1	4
Switch statements	5	1	4
Total	100	73	27

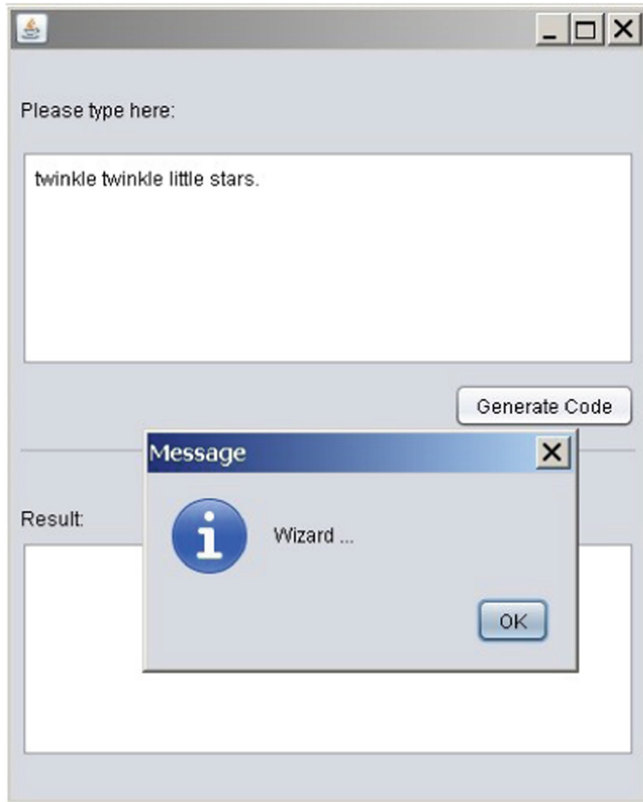

Fig. 9. Assistant wizard invoke

2/10, where is actual formula should be 2 * 10). Similarly the word do, which means do something whereas in computers sciences this word is used for loop. Most of the errors were observed in conditional statements especially in multi branching (switch statement). It is worth to mention that the proper data dictionary development and ambiguity are the core obstacle in programming with natural language in general. After implementation it can be said that there are three different situational cases can occur with the system that are:

Best Case
When the system produces output smoothly for provided input.
Average Case When the system initiate assistant wizard for provided input.
Worst Case
When the assistant wizard is not able to help user and directs him to application layer to provide more clear input. Figures 12, 13 and 14 shows best, average and worst case scenario respectively.

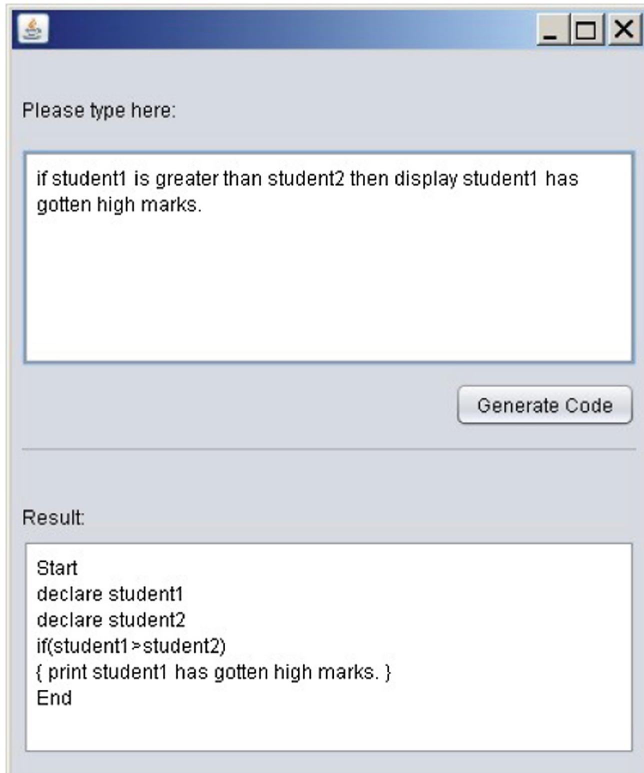

Fig. 10. Conditional statement result

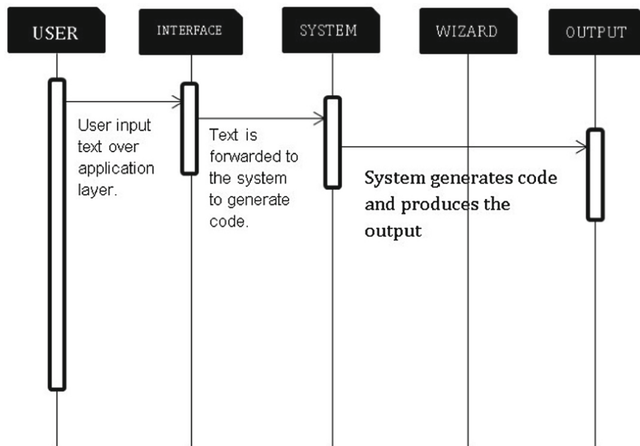

Fig. 11. Best case

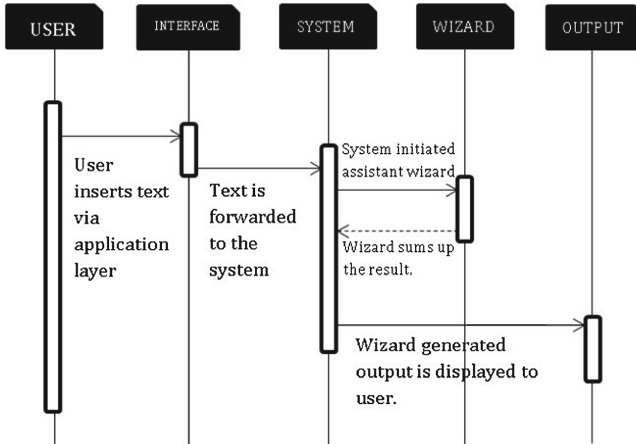

Fig. 12. Average case

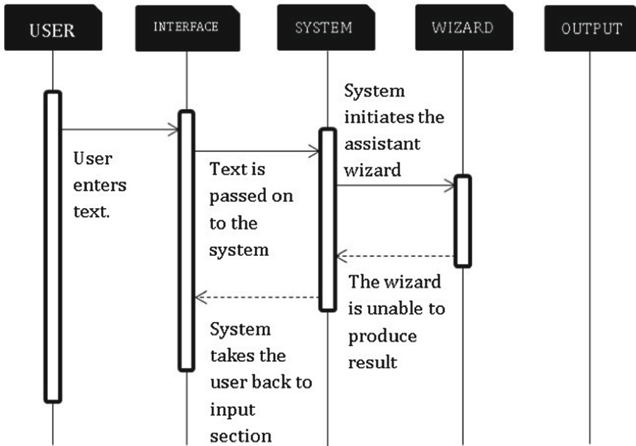

Fig. 13. Worst case

6 Conclusion

Code generation from natural language text should be considered as big system. This research work is a small contribution toward making such big system possible. The initial prototype test provided promising results at minute level however it is not claimed that the proposed model is absolute solution. It is intended to put more efforts in future in order to make it more reliable system. Moreover in future, more work need to tackle down ambiguity issue; proper assistant wizard and programming language code generation from pseudo code is recommended.

References

1. Stefik, A., Siebert, S.: An empirical investigation into programming language syntax. ACM Trans. Comput. Educ. (TOCE) **13**(4), 19 (2013)
2. Lieberman, H., Liu, H.: Feasibility studies for programming in natural language. In: Lieberman, H., Paternò, F., Wulf, V. (eds.) End User Development, vol. 9, pp. 459–473. Springer, Heidelberg (2006). https://doi.org/10.1007/1-4020-5386-X_20
3. Winograd, T.: Understanding natural language. Cogn. Psychol. **3**(1), 1–191 (1972)
4. Deeptimahanti, D.K., Sanyal, R.: Semi-automatic generation of UML models from natural language requirements. In: Proceedings of the 4th India Software Engineering Conference, pp. 165–174. ACM (2011)
5. Abirami, S., Shankari, G., Akshaya, S., Sithika, M.: Conceptual modeling of non-functional requirements from natural language text. In: Jain, L.C., Behera, H.S., Mandal, J.K., Mohapatra, D.P. (eds.) Computational Intelligence in Data Mining - Volume 3. SIST, vol. 33, pp. 1–11. Springer, New Delhi (2015). https://doi.org/10.1007/978-81-322-2202-6_1
6. Norouzifard, M., Davarpanah, S., Shenassa, M., et al.: Using natural language processing in order to create SQL queries. In: 2008 International Conference on Computer and Communication Engineering, pp. 600–604. IEEE (2008)
7. Androutsopoulos, I., Ritchie, G.D., Thanisch, P.: Natural language interfaces to databases-an introduction. Nat. Lang. Eng. **1**(1), 29–81 (1995)
8. Price, D., Rilofff, E., Zachary, J., Harvey, B.: Naturaljava: a natural language interface for programming in Java. In: Proceedings of the 5th International Conference on Intelligent User Interfaces, pp. 207–211. ACM (2000)
9. Vadas, D., Curran, J.R.: Programming with unrestricted natural language. In: Proceedings of the Australasian Language Technology Workshop, pp. 191–199 (2005)
10. Steedman, M.: The Syntactic Process, vol. 24. MIT Press, Cambridge (2000)
11. Nadkarni, S., Panchmatia, P., Karwa, T., Kurhade, S.: Semi natural language algorithm to programming language interpreter. In: 2016 International Conference on Advances in Human Machine Interaction (HMI), pp. 1–4. IEEE (2016)
12. Thomas, J., Antony, P.J., Balapradeep, K.N., Mithun, K.D., Maiya, N.: Natural language compiler for English and Dravidian languages. In: Shetty, N.R., Prasad, N.H., Nalini, N. (eds.) Emerging Research in Computing, Information, Communication and Applications, pp. 313–323. Springer, New Delhi (2015). https://doi.org/10.1007/978-81-322-2550-8_31
13. Anjali, M.K., Anto, P.B.: Ambiguities in natural language processing. Int. J. Innov. Res. Comput. Commun. Eng. 392–394 (2014)
14. Sag, I.A., Baldwin, T., Bond, F., Copestake, A., Flickinger, D.: Multiword expressions: a pain in the neck for NLP. In: Gelbukh, A. (ed.) CICLing 2002. LNCS, vol. 2276, pp. 1–15. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45715-1_1