A Compiler with a Different Computational Framework of Floating Point Arithmetic and Variable Length of Datatype

Somsubhra Gupta[1], Sabyasachi Ghosh[2], Pradipta Ghosh[3]

[1]Department of Information Technology,
JIS College of Engineering, Kalyani-741235, India,
E-mail: gsomsubhra@gmail.com
[2&3]Department of Computer Application
JIS College of Engineering, Kalyani-741235, India,
[2]E-mail: sabyamca007@gmail.com
[3]E-mail: pradiptaghosh1983@gmail.com

Abstract— **Programming Language Processors [1] are designed to convert a high level language written source code to equivalent object code. There are variations in them, viz. interpreter, compiler etc. based on the way they accomplish the task without changing its nature. The computational complexity in developing an application corresponding to a real-life problem is inherent in the code. Intrinsically, compilers have the privilege to test the quality of the code developed to solve the problem. This can be achieved during code optimization phase of the development where compiler can estimate the amount of redundancy or unnecessary expansion in code level after appropriate syntax and semantic analysis. These can occur for several reasons. Prominent reasons are the developers' constructed algorithm and the working framework of the compiler. The basic idea in the presented approach is to provide a framework of floating point arithmetic which is principal component of the design, primarily responsible for any numeric computation and is different from the standard practice of mantissa-exponent scheme [2]. The objective is to facilitate a broader perspective of development by allowing variable length of data type to the developer. The motivation behind this work is to facilitate identification of the source code under the interdisciplinary domain of Language Processor and Computational Intelligence. The proposed work can be further extended to analyze the trend of the code generated by the developer for incorporation of auto-debugger that will aid in getting rid of syntax errors in the source code.**

Index Terms— **Floating-point arithmetic, Compiler Design, Sharp Compiler, Machine Language.**

I. INTRODUCTION

A compiler is a set of computer programs that translates high level programming language written source code into system dependent object code. From a broader viewpoint, a compiler is communication medium that translates source language into target language. Compiler takes as input a source code that is written in high level language and creates an executable code in binary format in phases with the help of other component namely assembler, loader, linker and debugger. These phases are considered to be parts of Analysis-Synthesis phase [3]. The analysis phase starts with token or program element identification called as lexical analysis. This is followed respectively by syntax and semantic analysis [4] where grammatical correctness and collective meaning of each and every sentence written in the source code are analyzed according to the rule-base of the compiler. Symbol table manager and error handler are two supporting phases active throughout the analysis-synthesis phase to assist. The synthesis part deals mainly with code optimization and final object code generation.

The design of the compiler depends on the programming paradigm [5] they correspond, namely structured, procedure oriented, object oriented etc. Design of the compiler irrespective of compilation procedure can vary depending on the programming paradigm [6] they correspond. E.g. two-stage compilation in both C and C++, although one provides procedure oriented paradigm and the other provides object oriented paradigm of programming. The variation in design is incorporated into them so as to identify application designed using top-down and bottom-up approach. Compilers are identified according to their paradigm just as Assembly language [7] is identified as a low level language and PROLOG is identified to suit logic or invariant programming paradigm.

It was Von Neumann and his team, who innovate high speed digital computers that were introduced as a mass – scale storing and calculating device during 1940-46. But efficient language processing on that architecture was still at an early stage. The necessity, of expressing group of instructions using grammatically correct and specific notation, introduces the concept of language processor or compiler, theoretically. The machine level language is the lowest form of computer language introduced as the first programming language during the same time. Later, Assembly language is introduced as an extension to machine level language which incorporates mnemonics [8] instead of symbolic instructions. In spite of its capability to solve real

.

life problems, it was never really accepted beyond system programming level due to its complex-in nature type.

From its inception, due to its potential to contribute in large scale development, different aspects of compiler construct were investigated by various researchers in the field. Aho, Ullman, Scott [3, 9] are the leading researchers on the field focusing their investigation in parsing and syntax analysis phase of the compiler. Various forms of procedure ranging from subroutine to function and their introduction in Compiler were studied in [10]. Programming language pragmatics [5] was studied by Scott, Pratt et al.[11]. They have worked on theory of computation aspects and semantic analysis to capture collective meaning of tokens in the source code expressed in a programming language.

Design of programming language is one of the most prevalent art forms in the programming world. The number of programming languages proposed and designed is very high. During 1956-61, some prominent high level languages are introduced viz. FORTRAN, PASCAL, and COBOL with more enriched set of language features such as structured control construct, nested statements, blocks and procedures, files in addition to whatever available early. Later on, [12] introduced a procedural programming language named as C which emerged as one of the most powerful general purpose language. C is designed to provide procedure oriented paradigm of programming. But due to the supremacy of bottom up approaches in most of the real life problems, development of language processor with object oriented feature became evident. Consequently, C++ was introduced [13] during 1977 that incorporate many common object oriented features. By that time ADA was introduced, as an extension to PASCAL [14], which is highly structured and supports top-down development through 'package' subunits. Also it incorporates some object base feature. On the other hand, interactive programming features are introduced already by some compilers viz. LISP, PROLOG. These are classified among various generations of programming languages. Later on by converging from object oriented technology to object technology, with the aspiration of providing broad feature-centric approaches and user friendly design interfaces to enhance usability, Oak transformed later to JAVA [15] was introduced by SUN and visual programming platforms were introduced by MICROSOFT. The compilers of new era mostly use widely accepted common framework of design aspects on various features. E.g. most of them support fixed length datatype except for a few. Similarly they follow the standard representation procedure. E.g. most of the compilers follow IEEE 754 standard and IEEE 854 for floating point representations [16].

New approach towards designing compiler and extending it to natural language processor has been investigated [17, 18] from its potential scope of application in real life. In the recent past, Gupta et al. [19, 20] studied the compiler from both natural language and computational linguistic aspect using propositional logic. The study has been extended under the imprecise nature of instruction using fuzzy logic in [21].

In the presented work that deals with design and development of a compiler named Sharp Compiler introduces variations on the length of the datatype it provides and also on floating point representation process. In this work the compilers introduces data type with variable length which means length of the variable can be specified by the source code developer in a specific multiple form of 8. Though in some early language namely COBOL this is introduced, however difference remains in their representation. The proposed work suggests different form of floating point arithmetic from the standard practice of mantissa-exponent form. These are discussed in the next sections.

II. METHODOLOGICAL ASPECTS

A. *Variable Length Datatype Concept*

Datatype specifies to the system what kind of data is being dealt with i.e. the nature of attribute of data. This involves setting restrictions regarding what are the values within its scope, and what operations may be performed on those data. Datatype contains two properties of data one is the type of data and another is the range of values associated to it. This is done during abstract datatype specification. This presented work emphasizes on higher range of values for particular datatype. Since the number of keywords is not increased to create wider range of datatype, so there remains the provision to do so by adding functionality in the assembler level without affecting the phases of compilation procedure namely parsing, syntactical analysis, symbol table generation etc [22, 23, 24].

Sharp Compiler supports three datatype namely nrl, real and str, and with specification of size at the time of declaration in order to have a rational utilization of memory space. The length of variable can be specified with the operator “@”; the value that will be written after the operator will be treated as the length of the variable and its range of values will be determined accordingly. E.g. the declaration statement ‘nrl var@16’ means ‘var’ is an integer type variable having the 16 bits of memory space.

The datatype **‘**nrl’ is used for the integer type number and ‘real’ is used for the floating point number of varied length in multiple of 8-bit. The datatype ‘nrl’ is designed to support 8-bit, 16-bit, and 32-bit length so far and for ‘real’ these are 16-bit and 32-bit length in the sense that they are extendable under the presented framework of the design. The datatype ‘str’ is used to store the string type value i.e. a finite sequence of characters with range of 8-bit and 16-bit.

B. *Number Storing and Arithmetic*

Since the datatype and the number representation arithmetic are inter-related, so maintaining IEEE 754 standard in our perception will add bit complexity in development of the compiler, hence a new data structure is introduced for storing and operating numbers.

In the presented work every number has three properties, first, is the augmented value including the sign bit, second one is the number of digits after decimal point and the last one is the total number of digits of the number excluding the decimal point and sign bit. Depending on those three properties a number is stored using an algorithm named as 'Number Structure', described in the algorithm section.

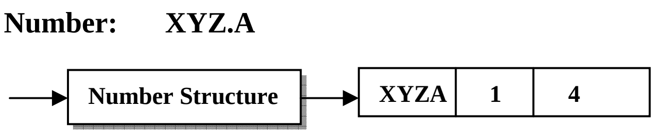


Figure I: Storing number (16-bit)

As illustrated in the Figure I, if a number in is in the form XYZ.A then the 1st field named as NUM of the structure will contains XYZA as augmented value. The 2nd field FC will count the number of digits after the decimal point i.e. 1 here. Last field ELEMI contains the total length of the number i.e. 4. Total pictorial view of the description is shown in the Figure I.

In the presented work, a single number system structure is sufficient to store both integer number and floating point number. The basic difference of two such storing is that the FC field of the structure is always zero for the integer type number and must have some positive value for any floating point number depending on the size of the variable. There are two type of precision, and for single precision operation the maximum length is 4-digit excluding decimal point and sign bit. For double precision operation the maximum length of a floating point number is 8-digit.

Apart from the basic mathematical operation [25] namely addition, subtraction, multiplication and division this compiler supports one more operator which is modulo division or remainder operator denoted by %. The operators %, /, * have the same as well as higher precedence than +, - and they are associated from left to right. The parentheses, which, in group, are used to determine the priority of an expression, have the highest precedence.

All numbers are stored in a structured format. Whenever any arithmetic operation will execute it is needed to form the original numbers from the structured format for the operation and need to store the result into structured format. To meet the requirement, two algorithms entitled 'synchro' and 'adjust' are implemented and describe below.

As in illustrated Figure II, let's consider an addition operation with two 16-bit floating point variables the result of which will be stored in another 16-bit variable. The values XXX.X and YY.YY are stored in the two variables. The result of this operation will be at least a 6-digit number. The resultant variable can hold a maximum of four digits; consequently a truncation of least significant digit from the number is needed.

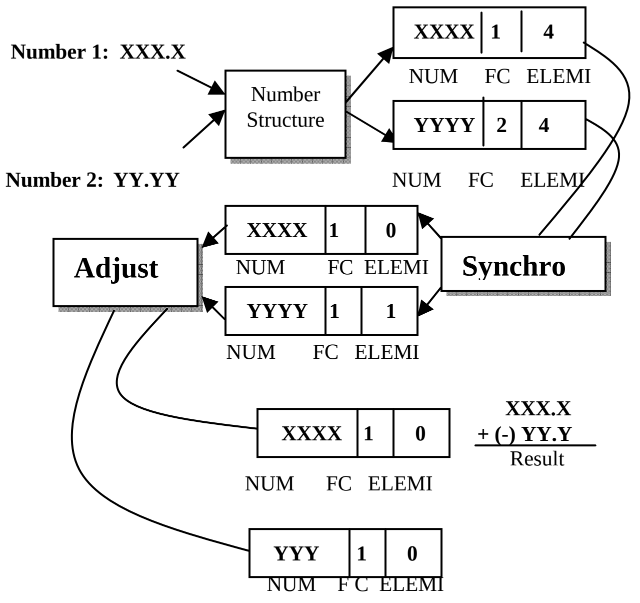


Figure II: Floating point operation (16-bit)

To recover the value loss up to a certain limit is possible by rounding off the number before truncation. Though it is illustrated symbolically in the aforementioned statement, the accuracy of the result produced does depend on the users' choice of the proper specification of size of the variables for the operation. Data won't be lost if same number of digit remains after decimal points in both the operands. If the difference in the number of digits after the decimal point of the two operands is high then the possibility of value loss is maximum. The following table illustrates a comparison statistically.

TABLE I
STATISTICAL COMPARISON FOR LENGTH VARIATION

Operation	Expected Result	System Result (in 16-bit)	Result incor-porating Rounding (in 16-bit)	System Result (in 32-bit)
123.4+0.1234	123.5234	123.5	123.5	123.5234
71.5*5.36	383.24	383.2	383.24	383.24
123.4-12.78	110.62	110.7	110.6	110.62
123.4+127.8	251.2	251.2	251.2	251.2
712/5	142.4	142.4	142.4	142.4
98765.43 +12.789	98778.219	-	-	98778.219
999.9+9.999	1009.899	1009.8	1009.9	1009.899
555.66666 +6.555555	562.222215	-	-	562.22221

From the observation it is clear that if the result of an operation of two 16-bit numbers is stored in a 32-bit variable, then there is no chance of value loss. However if the result is stored in a 16 bit variable, the value loss can be minimize using round off scheme so, it is preferable to store the result in a higher range so as to get the exact result. In the above table '-'entry signifies 'no result' as the input numbers are both of 32-bit which is not compatible for producing result in 16-bit.

III. ALGORITHMS IN SHARP COMPILER

Before working with the numbers it is important to deign a proper data structure. The presented work introduces floating point representation in a different way than IEEE 754 standard.

A. Algorithm: Number Structure

Number Structure (no):

The parameters NUM, FC, ELEMI are referred to in the Figure I. This work produces a different data structure to store floating point number. Here it is described step wise

Here 'no' is the formal argument and can be any real or integer

- The defined data structure has 3 components.
- First one named as NUM contains the augmented value of the 'no' ignoring the decimal point.
- Second field FC, contains the number of digits after decimal point.
- And the last field ELEMI contains the total number of digits.

The complete diagrammatical representation of the concept is illustrated in Figure I.

B. Algorithm: Binary Operations on two numbers

Before binary addition and subtraction operation, it is necessary to construct those two operands in such a way, it can produce the proper result, i.e. the two operands are adjusted depending on their FC value and modify the numbers, either by truncating some digits from or by appending zeros to one or both the operands making the FC value of the two operands identical.

These two following algorithms are designed to perform the aforementioned concept. The pictorial representations of the concept are described in the Figure II.

Algorithm: Synchro

Synchro (num1, ele1, num2, ele2, fc1, fc2)

Here num1 and num2 are two floating point numbers, ele1 and ele2 represent the no of digits of these two numbers respectively. And fc1 and fc2 are the number of digits after decimal point of num1 and num2 respectively. The variable 'len' can holds two integer values 4 or 8.

```
Step-1: initialize cmlimit=len and cmdis=0,cl=fc1,bl=fc2
Step-2: if (cl<bl) goto Step-24 otherwise go to Step-3
Step-3: if (bl==0) goto Step-4 otherwise go to Step- 6
Step-4: if (ele2<=cmlimit) goto Step- 6 otherwise goto Step-5
Step-5: assign ele2=0, ele1=fc1, fc1=0 and goto Step-29
Step-6: assign ch=ele2
Step-7: if (ch==cmlimit) go to Step-8otherwise goto Step- 9
Step-8: set cl=cl-bl, ele1=cl, fc1=bl, ele2=0 and goto Step- 29
Step-9: set cl=cl-bl, cl=cl+ele2
Step-10: if (cl<=cmlimit) goto Step- 15 else goto Step-11
Step-11: decrement cl as cl=cl-1
Step-12: increment cmdis as cmdis=cmdis+1
Step-13: set ch=fc1
Step-14: decrement ch as ch=ch-1 and set fc1=ch and goto
         Step-10
Step-15: set ch= cmdis
Step-16: ele1=ch, ch=fc1,cl=fc2
Step-17: set ch=ch-cl
Step-18: initialize ax=1
Step-19: if (ch==0) go to Step-22 otherwise goto Step-20
Step-20: set ax=ax*10
Step-21: decrement ch as ch=ch-1 and goto Step-19
Step-22: set bx=num2
Step-23: set ax=ax*bx, num2=ax, fc2=fc1, ele2=0 and goto
         Step- 29
Step-24: if (cl==0) go to Step-23 otherwise goto Step-28
Step-25: set al=ele1
Step-26: if(al<=cmlimit)go to Step- 28 otherwise goto Step-27
Step-27: ele1=0,al=fc2,fc2=0ele2=al and go to Step-29
Step-28: Swap (fc1, fc2), Swap (ele1, ele2), Swap (num1, num2), Set flag=true and goto Step-3
Step-29: if (flag==true) {Swap (fc1,fc2),Swap(ele1,ele2), Swap(num1,num2)} and goto Step- 30 otherwise goto Step-30
Step-30: stop.
```

Algorithm: Swap

Swap (num1,num2)

Here num1 and num2 are two integer numbers

```
Step-1: initialize locVar=0
Step-2: assign locVar=num1
Step-3: assign num1=num2
Step-4: assign num2=num1
Step-5: stop
```

Algorithm: Adjust

Adjust (num1, ele1, num2, ele2)

Here num1 and num2 are two floating point numbers, ele1 and ele2 represent the number of digits of these two numbers respectively.

```
Step-1: initialize cmsign=0
Step-2: if (num1<0) set num1= -num1 and cmsign=1
Step-3: repeat Step 4 while ele1! =0
Step-4: set num1=num1/10 and decrement ele1 as ele1=ele1-1
Step-5: if (cmsign! =1) goto Step-6 otherwise goto Step-8
Step-6: if (num2>0) goto Step-7 otherwise goto Step-10
Step-7: repeat Step-11 while ele2! =0
Step-8: set num1= -num1 and cmsign=0
Step-9: if (num2>0) goto Step-7 otherwise goto Step-10
Step-10: set num2=-num2 and cmsign=1and goto Step-7
Step-11: set num2=num2/10 and decrement ele2 as ele2=ele2-1
Step-12: if (cmsign==1) set num2= -num2 and cmsign=0
Step-13: stop.
```

IV. SYSTEM CONTROL FLOW

The flow of control of the proposed compiler is presented in the following Figure III.

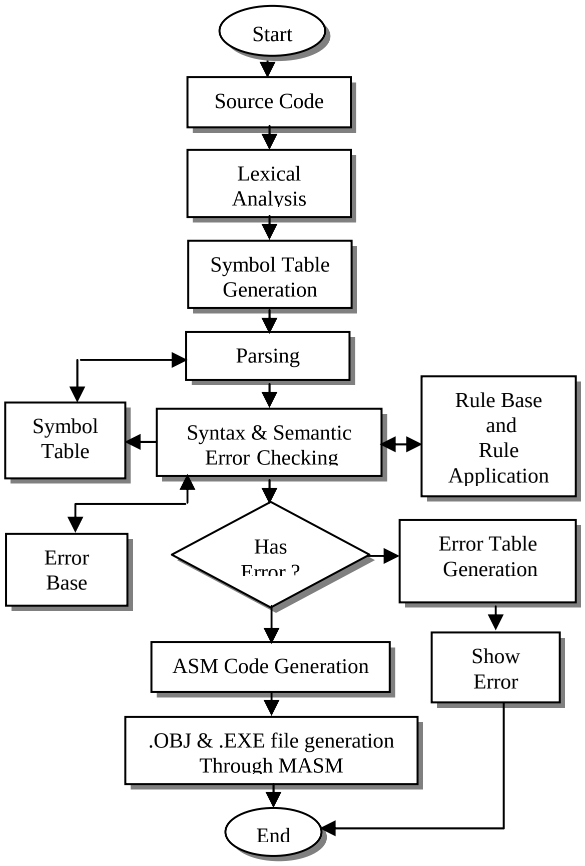
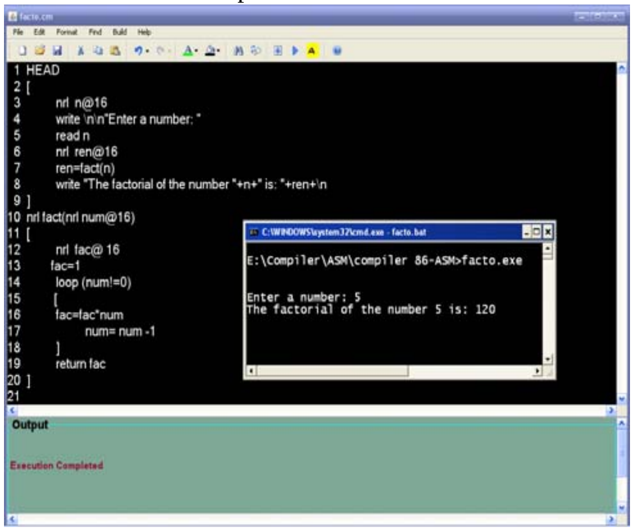


Figure III: System control flow

V. CASE STUDY

The core objective of the work is illustrated part wise below

- The system provides a developer friendly GUI where one has to write the source code according to the Sharp Compiler (SC) specific syntax.
- The entire source code is scanned line-wise and the parsed tree is generated with the help of system specific tokens from which the symbol table manager is constructed.
- The system consults with the predefined rule base and symbol table manager to do the syntax and semantic error checking for finding out the grammatical errors.
- Then a native assembler (MASM 6.1) specific assembly code is generated corresponding to the error free source code.
- In a process of, the Sharp Compiler uses native assembler to produce the result.
- In its own editor provided by the Sharp Compiler, the source code created by the developer is stored in .cm form or an already existing document can be opened. It facilitates developer to write and save the written document to compile and execute it.



```
HEAD
[
      nrl  n@16
      write \n\n"Enter a number: "
      read n
      nrl  ren@16
      ren=fact(n)
      write "The factorial of the number "+n+" is: "+ren+\n
]
nrl fact(nrl num@16)
[
      nrl  fac@ 16
     fac=1
      loop (num!=0)
      [
      fac=fac*num
            num= num -1
      ]
      return fac
]
```

```
C:\WINDOWS\system32\cmd.exe - facto.bat

E:\Compiler\ASM\compiler 86-ASM>facto.exe

Enter a number: 5
The factorial of the number 5 is: 120
```

```
Execution Completed
```

Figure IV: Sample source code in Sharp editor

It is to be observed that the presented compiler 'Sharp' is designed to convert high level language to assembly language using JAVA SE 1.6, and generated ASM code is compiled and executed by MASM 6.1. One sample code conversion is illustrated in the Table II. The code conversion deals with high level and purely object oriented programming paradigm to low level systems programming paradigm [26] during source to assembler conversion using JAVA. It's comparatively a new branch of study as compared to previously designed compiler and not enough literatures are available here. Very few literatures are available in constructing Compiler using conversion from High level language JAVA to Assembly, so the correlated study [27] are improvised with the needs and desires of the presented work.

The Table II describes the ASM code corresponding to the declaration part of the program 'factorial' shown in the Figure IV. The conception of the number storing are describing in the Section 2.2 and the in the Figure I is practically represented here.

TABLE II
ASSEMBLER CODE DECLARATION PART

1	2
.MODEL SMALL	WriteBuff BYTE 256 dup(' ')
.STACK 64	CMtemp32_arr CM32bit 5
PUTC MACRO char	DUP({?})
PUSH AX	CMtemp_arr CM16bit 5 DUP({?})
MOV AL, char	CMtemp8_arr CM8bit 5 DUP({?})
MOV AH, 0Eh	CMVAR16bit1 CM16bit <>
INT 10h	CMVAR16bit2 CM16bit <>
POP AX	CMVAR8bit1 CM8bit <>
ENDM	CMVAR8bit2 CM8bit <>
.DATA	CMVAR32bit1 CM32bit <>
CM32bit STRUCT	CMVAR32bit2 CM32bit <>
FC DB ?	CMVARDIS32bit2CM32bit <>
ELEMI DB ?	CMSTRBUFLEFTBRANCH16DB 512
NUM32 DD ?	dup(' ')
CM32bit ENDS	CMSTRBUFRIGHTBRANCH16DB
CM8bit STRUCT	512 dup(' ')
ELEMI DB ?	CMSTRBUF8 DB 256 dup(' ')

```
NUM8    DB    ?
CM8bit   ENDS
CM16bit STRUCT
FC DB    ?
NUM DW    ?
ELEMI   DB    ?
CM16bit ENDS
CMW411    DB'Enter
anumber:'
CMW827    DB'The factorial
of the number '"
CMW833    DB ' is: '"
```

```
CMSTRBUF16     DB 512 dup(' ')
CMSTRBUF32     DB 1024 dup(' ')
v4 CM16bit     <>
v16 CM16bit     <>
v44 CM16bit     <>
v50 CM16bit     <>
CMresult1    CM16bit    <>
CMresult2    CM16bit    <>
CM8result1 CM8bit    <>
CM8result2 CM8bit    <>
.386
.CODE
```

VI. CONCLUSION

The presented work that introduces a new compiler 'Sharp' with different floating point arithmetic and customized length of variables. The introduced syntax in the compiler is similar to that of any other high level language. It is structured and built to support the procedure oriented programming paradigm at application end. However during the development process the other paradigms of programming namely object oriented and low-level is rigorously investigated. This is due to the filed of implementation where programming language JAVA is selected for compiler design and implementation. JAVA is an object oriented language. So, at the language processor development end, object-orientation is there but it supports procedure oriented applications only. Again JAVA to Assembly conversion and incorporation of assembler provides the opportunity of investigation at low-level programming.

The presented compiler is planned to be expanded further to enable auto-debugger. The idea is to trace the probable errors from its error-handler. If the plausible debugged code is unique then the embedded auto-debugger will rectify the mistakes at once. In the case of variation in the type of error, following cases may arise. Most of these cases occur when a mistake can have non-unique corrected form.

In case of any run time error the compiler will prompt the programmer with the error and then provide him with a correct solution, and thereby helping him to modify his code according to his wish. This is much closer to manual debugging, however, the complexity at user end is reduced as user-don't have to think about location and type of error. This will be employed with rule-based way under the framework of computational intelligence.

The alternative is to mine the pattern of error which will open a different dimension of research. This requires Data Mining to identify trend of programming in user of different levels. This is followed by training of the system using Neural Network to develop working knowledge base about its user of various category. And then finally test with number of user developed source code to set various parametric values for successful identification of debugged form. This part of investigation is planned to be under the domain of Soft Computing.

ACKNOWLEDGMENT

The authors are grateful to the Organizing Chairs, ACVIT 2009 and the anonymous referees for their kind suggestion for improvement of the quality of presentation. The authors are deeply indebted to the Authority and Management of JIS College of Engineering for continuous encouragement during the course of the work.

REFERENCES

[1] A.V. Aho, and J.D. Ullman, *Principles of Compiler design*, Reading, Mass: Addison Wesley,1977

[2] D. Goldberg, "*Computer arithmetic. In Computer Architecture: A Quantitative Approach*", by J. L. Hennessy and David A. Patterson, pp. A1-A77, San Francisco, 2nd Ed., Morgan Kaufmann,1996.

[3] A.V. Aho, M.S. Lam, Ullman, J.D. Ullman and R. Sethi, *Compilers: Principles, Techniques and Tools*, 2nd Ed., Pearson Ed., 2006

[4] D.A. Watt, "*The parsing problem for affix grammars*". Acta Infomatica, vol. 8(1), 1977, pp.1-20

[5] M.L. Scott, *Programming Language Pragmatics*, Morgan Kaufmann Publishers, 2000

[6] A.B. Tucker and R.E. Noonan, *Programming Languages: Principles and Paradigms* ,2nd Ed., McGraw Hill, 2007

[7] P. Abel, IBM PC Assembly Language and Programming, 5th Ed., Pearson Ed., 2001

[8] P. Norton and J. Socha, *Assembly Language for the PC*, 3rd Ed., Prentice Hall of India, 1996

[9] M.M. Michael and M.L. Scott, "*Simple, fast, and practical non-blocking and blocking concurrent queue algorithms*," in Proceedings of the Fifteenth Annual ACM Symposium on principles of Distributed Computing, pp.267-275, Philadelphia, PA, May 1996

[10] A.V. Aho, and J.D. Ullman, *The Theory of Parsing, Translation and Compiling*, Vol.1, Parsing, Englewood Cliffs, N.J: Prentice-Hall, 1972

[11] T.W. Pratt, *Design and Implementation of Programming Language*, Englewood Cliffs, N.J: Prentice-Hall, 1975

[12] B.W. Kernighan and D.M. Ritchie, *The C Programming Language*, Englewood Cliffs, N.J: Prentice-Hall 1978

[13] B.J. Stroustrup, *C++ Programming Language*, 3rd Ed., Pearson Ed., 2000

[14] N. Wirth, *The Programming Language PASCAL (Revised Report)*, Technical Report No.5, Eidgenossische Technische Hochschule, Zurich, 1973

[15] H. Schildt, *JAVA 2: The Complete Reference*, 5th Ed. McGraw Hill, 2005

[16] D. Goldberg, "What Every Computer Scientist Should Know About Floating-Point Arithmetic", Computing surveys (March, 1991), ACM

[17] M. Lebowitz, "Generalization from natural language text", Cognitive Science, vol. 7(1), 1983,pp.1-40

[18] S.M. Sheiber, "A uniform architecture for parsing and generation" In proceedings of 12th International Conference on Computational Linguistics, 1988, pp. 614-619.

[19] S. Gupta, "Smart Compiler: A Computational Linguistic Approach to Compile and Execute Pseudo Code" in proceedings of 5th International Conference on Information Science, Technology and Management (CISTM'07), 2007, pp. 18, 1-10.

[20] S. Gupta, S. Chakraborty, "Smart Compiler: A Natural Language Processing Approach to Compile and Execute Pseudo Code" in Proceedings of International Conference on Computer Vision and Information Technology (ACVIT'07), 2007, pp. 383-390.

[21] B.B. Pal, S. Gupta, "Fuzzy Linguistic approach to design Pseudo Code Compiler in Multiobjective Decision Making Framework" at 2nd national Conference on Recent Trends In Information Systems (ReTIS'08), 2008, pp. 143-148.

[22] A.V. Aho and M.J. Corasick, "Efficient string matching: an aid to bibliographic search" Comm. ACM vol. 18:6, 1975, pp.333-340

[23] A.V. Aho, B.W. Kernighan and P.J. Weinberger, AWK: A Pattern Scanning and Processing Language, Bell telephone Labs, Murray Hill, NJ, 2nd Ed., Sept 1978

[24] J.P. Trembly and P.G. Sorensen, *The theory and practice of Compiler Writing*, McGraw Hill Inc., 1985

[25] R.W. Floyd, "*Syntactic analysis and operator precedence*", J. ACM, vol.10:3, 1963, pp.316-333.

[26] J.J. Donovan, *Systems Programming*, McGraw Hill (Computer Science series), 1972

[27] A. Holub, *Compiler Design in C*, Prentice Hall of India, 2001