# Presenting JECA: A Java Error Correcting Algorithm for the Java Intelligent Tutoring System

Edward R. Sykes
*School of Applied Computing and Engineering Sciences, Sheridan College, 1430 Trafalgar Road, Oakville, Ont., Canada, L6H 2L1
email: ed.sykes@sheridanc.on.ca
phone: (905) 845-9430
fax: (905) 815-4035*

Franya Franek
*Department of Computing and Software, Faculty of Science, McMaster University, 1280 Main Street W., Hamilton, Ont., Canada, L8S 4L8
e-mail: franek@mcmaster.ca
phone: (905) 525-9140*

## Abstract

*Error recovery in compiler design and construction is a well-known area of Computer Science. Traditionally, the compiler's responsibility has been to identify all possible errors in one pass of the source code in as short a period of real time as possible. However, in certain situations, it is more desirable to have the compiler act 'intelligently' by making 'intelligent' code changes and by offering suggestions to the author of the source program. This research paper examines error recovery in a specific context involving small Java programs. Furthermore, this paper presents JECA (Java Error Correction Algorithm), a practical algorithm for a compiler that error corrects by intelligently changing code, and identifies errors more clearly than other current-day compilers. The ultimate goal of this research is to provide a foundation for the Java Intelligent Tutoring System (JITS) currently being field-tested.*

## Key Words

Intelligent Tutoring Systems, Computational Intelligence, Web Technologies, e-Learning.

## 1. Introduction

Today's compilers perform error recovery but still maintain a high level of terse error messages as feedback. These error recovery mechanisms act in much the same way as traditional systems in that they attempt to identify as many of the errors in the program as possible in the shortest amount of time. For instance, the default philosophy for error correction implemented in many compilers (e.g., C, C++, Java, Pascal, Turing, etc.) is to:

i) report the presence of errors accurately;
ii) recover from each error quickly in order to detect subsequent errors; and
iii) not significantly slow down the processing of correct programs.

However, in certain circumstances, as in learning to program, it is more desirable to have the compiler act 'intelligently' and make 'intelligent' changes or suggestions to the author of the source program. This research paper examines error recovery in a specific context involving small Java programs. A review of various tools is presented including JFlex, CUP, and JavaCC [1]. Furthermore, this paper presents JECA – a practical algorithm for a compiler that error corrects by intelligently changing code and identifies errors more clearly than other current-day compilers. The goals of the proposed system are to:

i) intelligently recognize the 'intent' of the student;
ii) analyze the student's code submission;
iii) 'auto-correct' where appropriate (e.g., converting "While" into the keyword "while", "forr" into "for", etc.);
iv) learn individual student's misconceptions, and categorizes the types of errors he/she make;
v) produce a 'modified code' that will compile (or bring the code closer to a state of successful compilation),
vi) produce a 'modified code' that will meet the program specifications (or bring the code closer to meeting program specifications); and
vii) prompt the student programmer for information when necessary via well-defined hint support structures.

The remainder of this paper provides a closer look at how these goals are achieved in JECA. In order to support the rationale for JECA, an investigation of appropriate tools including JFlex, CUP, and JavaCC. These tools are discussed in relation to their error recovery capabilities and potential to implement specific error recovery algorithms. The JECA implementation is presented with its supporting algorithms. Lastly, the integration of JECA with the Java Intelligent Tutoring System is discussed including specific examples using the embedded hint generation infrastructure. The purpose of JECA is to give clear and helpful feedback to the student. In this way, JITS supports students to be able to learn programming better and more enjoyably.

## 2. The Implementation: The Java Error Correction Algorithm (JECA)

JECA is supported by two distinct components. The first component involves scrutinizing the identifiers that the scanner has tokenized by comparing them to keywords and to currently validated identifiers. The second component has the parser perform a rigorous deep level error recovery technique implemented by a variation on the Burke-Fisher Error Recovery algorithm [2]. This algorithm is explained in greater depth in the following sections.

## 2.1 First Component of JECA: Error Recovery in the TokenManager (Scanner)

It is sometimes desirable to change what the scanner has interpreted to a single keyword. For example, suppose the beginner programmer submitted the following code:

```
public class Test {
  public static void main() {
    Int sum = 0;
    For (iint i=0; i<=10; i++)
      sum = sum + i;
    System.out.println("Sum is:" + sum);
  }
}
```

There are 3 distinct syntax errors. The "`Int sum=0;`" statement, the "`For`", and the "`iint`". It is desirable to present the appropriate information to the student programmer in a way that is both supportive and direct. In this example, the student mistakes the "`Int`" and "`For`" for the keywords "`int`" and "`for`" respectively. A typical compiler will produce the following:

```
Test.java:5: ')' expected
      For (iint i=0; i <=10;   i++ )
                ^
Test.java:5: not a statement
      For (iint i=0; i <=10;   i++ )
                       ^
Test.java:5: ';' expected
      For (iint i=0; i <=10;   i++ )
                                   ^
3 errors
```

The error recovery algorithm presented in this paper, JECA, attempts to understand the 'intent' behind the student's program and by prompting the student, and behind-the-scenes, modifies the submitted program as follows:

```
public class Test {
 public static void main(String args []){
   int sum = 0;
   for (int i=0; i <=10;   i++ )
      sum = sum + i;
   System.out.println("Sum is:" + sum);
 }
}
```

generating the anticipated result:

**`Sum is:55`**

The student will receive prompts for each 'assumption' the JECA intent recognition module is performing. For example, on encountering the 'Int' in line 3, a message will be produced "I found an 'Int'. Should I replace it with 'int' ? (y/n)" In this fashion, the user of the system is fully aware of all changes that are taken place on the submitted code. This philosophy is different from other compiler designs which make changes to the source program without notifying the user [3, 4]. A supporting mechanism used to do this is depicted in figure 4.

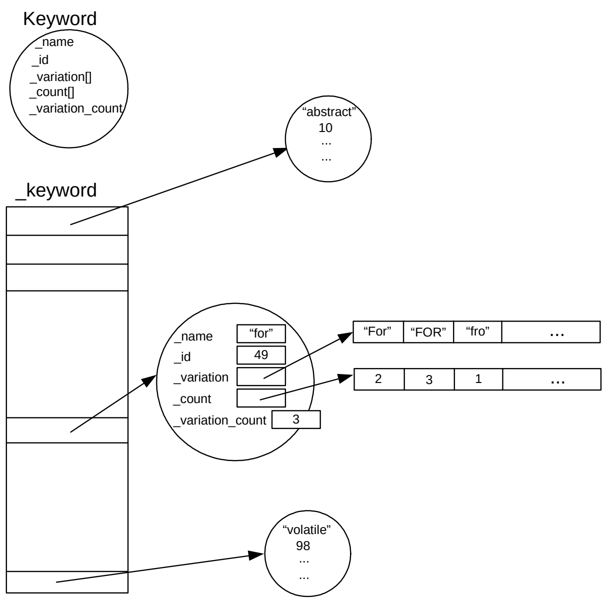


Figure 4. Keyword object and _keyword data structure

A Keyword object houses all attributes and functionality associated with a keyword in the language. It contains the name of the keyword (i.e., String _name), the symbol table ID for the keyword (i.e., int _id), dynamically learned variations on the keyword (i.e., String _variation []), the number of times these corresponding variations have occurred (i.e., int _count []), and the total number of variations learned at this time (i.e., int _count). The Keyword object contains useful information that can be used for statistical analysis and capturing a representative model of the student of the system. By keeping track of the types of errors the student makes and the number of times these types of errors occur, the system is in a good state to offer meaningful feedback to assist the student to program better. Given the lexeme (the identifier to be validated as an identifier or as a keyword) the algorithm for this process is presented below:

```
loop
   i = 0
   go through the _keyword array
   extract the keyword name at position i
   d = Edit_Distance (lexeme to keyword)
   if (d <= THRESHOLD)
          add it to a refinement collection
      i++
end loop
perform refinement on refinement collection
```

JECA uses an additional object called 'BestMatch' to assist in refining the search for appropriate potential keyword matches. The refinement collection is a Java Collection of BestMatch objects which represents the best matches of all the keywords that are similar to the identifier in question. The refinement process proceeds and applies additional rules and constraints to narrow the number of BestMatches until it is determined that the identifier is indeed a valid identifier or should be converted into a keyword. Once this is determined, the TokenManager returns the appropriate Token to the parser. A figure of the BestMatch object is presented in figure 5.

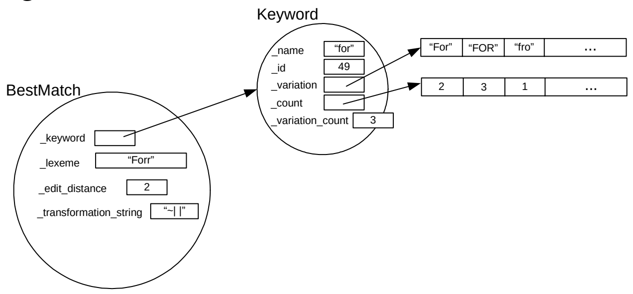


Figure 5. BestMatch object – used for the refinement process in determining an identifier or a keyword.

A member of the BestMatch object is _transformation_string. This member receives the value from the Edit_Distance algorithm. The Edit_Distance algorithm accepts two strings for comparison and determines the closeness of these strings by performing insertions, deletions, and character replacements [4]. Figure 6 depicts a transformation string given two strings "Forr" and "for".

```
Forr
~| |
fo-r
```

Figure 6. BestMatch member contains the Transformation string from Edit_Distance algorithm.

## 2.2 Second Component of JECA: Error Recovery in the Parser

JECA's parser component algorithm implementation is loosely based on the Burke-Fisher Error Recovery algorithm [2]. This algorithm exhaustively tries single token insertion, deletion or replacement at every point within k tokens before where the error occurs. In other words, k represents a window of tokens where the problem resides. Given N, representing the total number of tokens in the language, there are k+kN+kN possible deletions, insertions and substitutions within the k token window [2]. The k token window is kept on a queue. In this algorithm, all semantic actions must be delayed to prevent unwanted side effects until parse is validated [2].

The Burke-Fisher Error Recovery algorithm uses 2 stacks, *current* and *old*, and a *queue* of *k* tokens [2]. *old* stack contains all successfully parsed tokens so far. *current* stack contains potential tokens covering a window of the next *k* tokens. *old* stack and *queue* are used together to reparse string after replacement, deletion or insertion of single token into *queue.* Figures 7 and 8 depict an example using the Burke-Fisher error recovery algorithm.

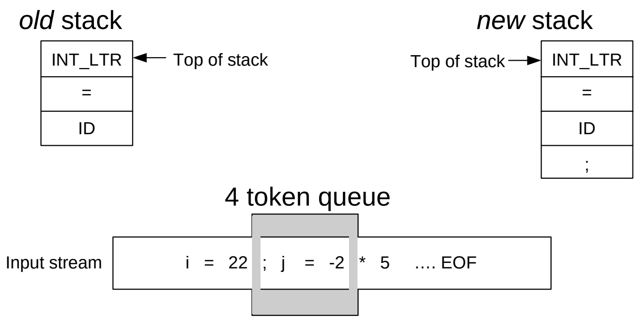


Figure 7. Burke-Fisher error correction algorithm with a 4-token queue in the middle of processing a statement production.

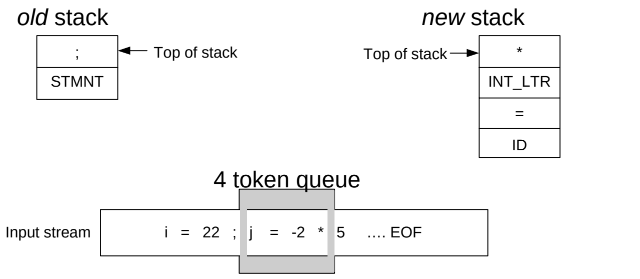


Figure 8. Burke-Fisher error correction algorithm with a 4-token queue completing the processing of a statement production and commencing a new production.

The proposed parser error recovery algorithm for JECA is similar in nature to the Burke-Fisher algorithm. However, there are some significant differences. First, since JECA is aimed at the beginner Java programmer, the size of the source program will always be very small (i.e., 50 lines of code or less). As a result, a Vector (i.e., `java.lang.Vector`) Abstract Data Type (ADT) is used to store the entire source program in memory. In this fashion, the tokens can be easily traversed and manipulated thus providing opportunities for greater analysis on the input program. Second, the Burke-Fisher algorithm delays semantic actions to prevent unwanted side effects. In JECA there are no semantic actions as would be expected in a typical compiler. In other words, unlike other compilers that generally produce assembler code, or 3-address code, the proposed algorithm's goal is to correct errors so that the parse will be as valid as possible. It does not have extensive semantic actions like other compilers. The output of the proposed algorithm is a modified source code that is intended to successfully parse by the standard 'javac' executable (i.e., Java compiler). The standard Java compiler will be invoked next to perform the translation from the modified source program to 3-address code. The third main difference between Burke-Fisher's algorithm and JECA's is that the student programmer will be asked for clarification during the error recovery session. So, instead of using Burke-Fisher's approach to exhaustively insert, replace, or delete tokens in a k-window token list, only the most probable tokens will be presented to the student programmer. As a result, the student has a significant degree of control over the recovery process. This is supported by an inner module which generates parse tree variations which are then tested against the parser and Java compiler. These variations are based on a number of considerations involving token replacement, deletion, insertions, and transpositions. A competition is arranged so that the parse trees that succeed in recognizing the most tokens in the source code are selected for further scrutiny. It then becomes a competition among the best trees to determine the appropriate course of action in terms of determining the specific hints issued for the student. Table 2 depicts this internal JECA functionality. Please note the student does not see these computations.

Table 2. Internal JECA parse tree permutations and competition for the selection of the best trees.

Given the following program:

```
1  public class Test {
2   public static void main(String args []){
3     iint sum = 0 ;
4     FOR ( Int i=0; i<10   i++ ) // missing ‘;’
5       sum = smu + i;
7   }
8 }
```

and submitting it to JECA will yield a `ParserException` stating:

```
Line 4 Column 30
Offending token: kind=>identifier, image=> “i”
Previous to Offending token:
kind=>integer_literal, image ==> “10”
```

The `ParserException` contains a list of expected tokens:

```
Expected ...
;
=
>
<
==
<=
>=
etc.
```

JECA takes this “expected” list, creates permutations on the base parse tree involving insertions, deletions, replacements, and transpositions, and then sets up the competition to determine the best tree…

```
Nothing compiled successfully...but here is the
best tree...

public class Test {
 public static void main(String args [] ) {
   int sum = 0 ;
   for ( int i = 0; i < 10; i++ )
      sum = smu + i;
   }
}
```

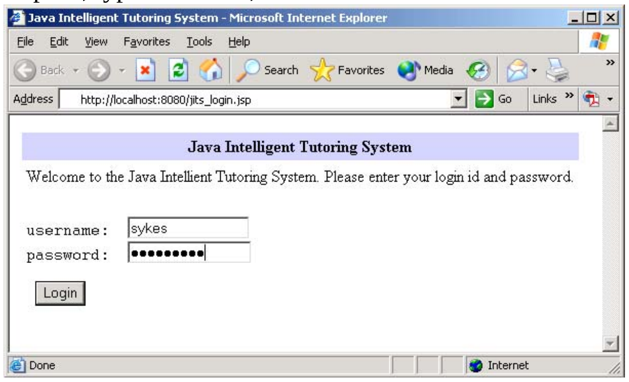
The fourth difference between the Burke-Fisher algorithm and JECA is that the parsing stops when it encounters a situation that it cannot satisfy the current production. The justification for this stems from the philosophy behind teaching beginning programmers [5, 6, 7]. It is important that the student programmer not become overwhelmed by the number of error messages typically produced by compilers when errors occur [8, 9]. Rather, it is more helpful to:

i) extract detailed information regarding the single error message and stop parsing;
ii) provide ***one*** clear and meaningful error message to the student; and
iii) encourage the student to make the correction [10].

## 3. Java Intelligent Tutoring System User Interface

The interface for computer-based programming tutors was given careful consideration during the design of the Java Intelligent Tutoring System (JITS). The user interface is based on a presentation format implemented in many popular Integrated Development Environments used by professional programmers (e.g., Visual Café, JDeveloper, JBuilder, etc.) [11]. The JITS login screen and user interfaces are shown in figure 9 and figure 10 respectively.

The student types in his/her solution in the Source Code Area and presses ‘Submit’. This invokes a call to the corresponding JavaBean representing the student. The code is then dispatched to JECA, which processes the submission and generates a set of appropriate hint objects. The student, at any time, may explicitly request a hint from JITS by pressing the Hint button, or view the solution by pressing the Solution button. The student may opt to select another problem, or quit the tutoring session at any time. Additionally, in support of metacognitive development, the student may view his or her own performance history (i.e., My Performance button). This displays a performance summary based on numerous statistics such as problems attempted, problems solved, number of attempts on a problem, problem difficulty, time elapsed, types of errors, etc.


Java Intelligent Tutoring System - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://localhost:8080/jits_login.jsp
Go Links
Done
Internet

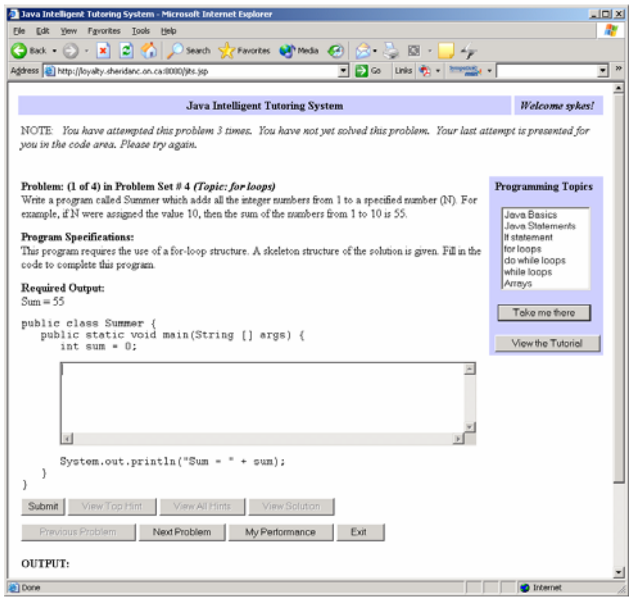
Java Intelligent Tutoring System

Welcome to the Java Intellient Tutoring System. Please enter your login id and password.

username: sykes
password: ●●●●●●●●●

Login

Figure 9. JITS login screen.



**Java Intelligent Tutoring System** *Welcome sykes!*

NOTE: *You have attempted this problem 3 times. You have not yet solved this problem. Your last attempt is presented for you in the code area. Please try again.*

**Problem: (1 of 4) in Problem Set # 4 *(Topic: for loops)***
Write a program called Summer which adds all the integer numbers from 1 to a specified number (N). For example, if N were assigned the value 10, then the sum of the numbers from 1 to 10 is 55.

**Program Specifications:**
This program requires the use of a for-loop structure. A skeleton structure of the solution is given. Fill in the code to complete this program.

**Required Output:**
Sum = 55

```
public class Summer {
   public static void main(String [] args) {
      int sum = 0;

         System.out.println("Sum = " + sum);
   }
}
```

**OUTPUT:**

Figure 10. JITS User Interface.

## 4. Java Intelligent Tutoring System Architecture

The JITS infrastructure supports the student via a browser accessing information from the tutor via an HTTP request/response process model. The processing is accomplished by JavaBeans™ within a servlet engine web server. The presentation layer uses JavaServer Pages™ technology which communicates to the bean representing the student and creates an XHTML page for the student's browser. During processing the bean gathers all the information about the student's code and submits it to JECA for processing. The infrastructure architecture uses a JDBC connection from the JavaBeans™ to an external database which stores and retrieves specific information about the student including student history and performance statistics.

The implemented architecture has numerous benefits [12]. It is scalable, platform-independent, and lightweight [12]. The student will never need to install software on his/her machine and will not need a high-speed network connection to use JITS. Other benefits include fast execution as all processing is done on the middle-tier web server, currently equipped with 4GB RAM and 2 Pentium-IV processors. The net result is a product that increases the accessibility for JITS to many students – a vital requirement for an equitable and successful educational product in today's Internet-ready community.

## 5. Hint Generation

An additional design consideration is the categories of hints that are generated by JECA for JITS. There are five types of hints that may be created as a result of an error from the student's code submission. They are:

```
KEYWORD_REPLACEMENT_HINT
EXTENDED_TYPE_REPLACEMENT_HINT
IDENTIFIER_REPLACEMENT_HINT
GRAMMATICAL_HINT
GENERAL_HINT
OTHER_TYPE_OF_HINT
```

Figure 11. Hint categories.

A `KEYWORD_REPLACEMENT_HINT` arises from a situation where the student typed in a suitably close representation to a Java keyword. For instance, if the student typed in '`Whiles`', this would be interpreted as the keyword '`while`'. An `EXTENDED_TYPE_REPLACEMENT_HINT` is when the student wrote '`Sting`' which will interpreted as '`String`' – the `java.lang.String` class. An `IDENTIFIER_REPLACEMENT_HINT` is used in the situation where a suitably close match to an existing identifier has been found. For example, consider the following snippet of code:

```
int my_int = 0;   // declaration
my_it = my_intt + 1; // and use
```

There would be two `IDENTIFIER_REPLACEMENT_HINT`s generated for this piece of code:

***Identifier Replacement Hint: Would you like me to replace "my_it" with "my_int"?***

***Identifier Replacement Hint: Would you like me to replace "my_intt" with "my_int"?***

A `GRAMMATICAL_HINT` is generated when the parser fails on a particular production in the Java grammar. Specific information regarding the error is recorded in the Hint object depicted in figure 8. The last two types of hint are `GENERAL_HINT` and `OTHER_TYPE_OF_HINT`. `GENERAL_HINT` is used in the situation when the student is far from the solution path and needs to be realigned with the program statement and program specifications for the posed problem. `OTHER_TYPE_OF_HINT` is reserved for future research.

There are a number of important pieces of information represented in a Hint object. The Hint object is depicted in figure 12. The _type member corresponds with one of the six types of categories of Hints currently supported in JECA. The _col and _line members specify where the error occurred. The _line_of_code and _error_pointer represent the source code and the exact location of where the error occurred. There are two tokens to assist in identifying where the error occurred in terms of the tokens. _offending_token represents the precise token the parser failed on, and _previous_to_offending_token represents the last successfully parsed token during parsing. The _hint member is a String summarizing the actual hint relying on the values of other data members in this object. It is intended to be used during the feedback process during student tutoring. The last member of the Hint class is the _confidence, which will be assigned an integer from 1 to 10. A confidence value of 1 indicates a high level of certainty indicating the suggested hint is correct and will bring the student closer to a compiled program. On the other hand, a confidence value of 10, indicates uncertainty on behalf of the hint generated. In these situations, the student will have to use their own judgment based on the detailed information provided to them by the Hint objects, namely the data members, _type, _col, _line, _line_of_code, _error_pointer, _offending_Token, and _previous_to_offending_Token.

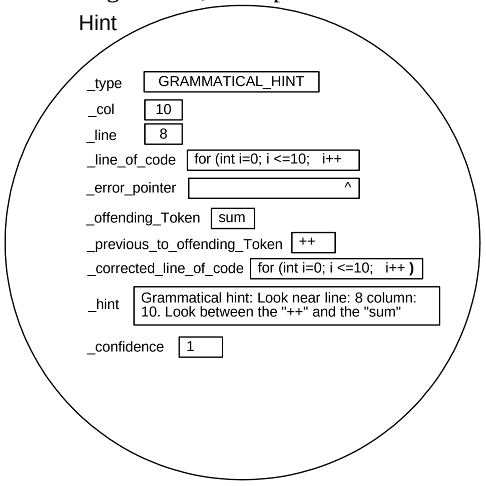

Hint
_type
GRAMMATICAL_HINT
_col
10
_line
8
_line_of_code
for (int i=0; i <=10;   i++
_error_pointer
^
_offending_Token
sum
_previous_to_offending_Token
++
_corrected_line_of_code
for (int i=0; i <=10;   i++ )
_hint
Grammatical hint: Look near line: 8 column: 10. Look between the "++" and the "sum"
_confidence
1

Figure 12.  A JECA Hint object representing a grammatical error.

An example follows to illustrate these design aspects of the proposed error correction algorithm.

Given the following source program:

```
public class Test {
 public static void main() {
   Int sum = 0;
   For (iint i=0; i<=10; i++
     sum = sum + i;
   System.out.println("Sum is:" + sum);
 }
}
```

Figure 13.  Arithmetic sum Java program with grammatical errors and syntax errors.

JECA would modify the program to:

```
public class Test {
 public static void main(String args []) {
   int sum = 0;
   for (int i=0; i <=10;   i++ )
      sum = sum + i ;
   System.out.println("Sum is:" + sum);
 }
}
```

Figure 14.  Internally corrected JECA source program for the arithmetic sum problem.

As a result, the following Hint objects would be created by JECA:

1) **Keyword replacement hint: Would you like me to replace "Int" with "int"?**
2) **Keyword replacement hint: Would you like me to replace "FOR" with "for"?**
3) **Keyword replacement hint: Would you like me to replace "iint" with "int"?**
4) **Grammatical hint: Look near line: 8 column: 10. Look between the "++" and the "sum"**

The following section depicts how the Hint objects are used in a typical dialog between JITS (via the supporting JECA module) and the student programmer.  Using the example presented in figure 15, focusing only on the area where the student enters code in the "source code area" (see figures 10 and 15), table 3 presents the dialogue between JITS and the student.

```
public class Summer {
 public static void main(String[] args) {

   int sum = 0;

      // source code area:
      // student writes code here

    System.out.println("Sum = " + sum);
 }
}
```

Figure 15.  Arithmetic sum Java program with supporting framework to focus the student on the task at hand.

Table 3.  Hint objects utilization and typical dialogue between JITS and the student.

Student's submission:

```
For (intt i = 1; i <= 10;  i++ {
      sum = smu + i;
}
```

**JITS:  Would you like me to replace "For" with "for"?**
*(Keyword replacement hint)*
**Student:**  Clicks Yes, or changes the code manually.
**Resulting code:**

```
for (intt i = 1; i <= 10;  i++ {
      sum = smu + i;
}
```

**JITS:  Would you like me to replace "Int" with "int"?**
*(Keyword replacement hint)*
**Student:**  Clicks Yes, or changes the code manually.
**Resulting code:**

```
for (int i = 1; i <= 10;  i++ {
      sum = smu + i;
}
```

**JITS:**  Look near line: 4 column: 37. Look between the "++" and the "{"  *(Grammatical hint)*
**JITS elaborates:**

**HINT STRING :**

```
 for ( int i=0; i<10;  i++   {
                             ^
```

**CORRECTED CODE:**

```
for ( int i=0; i<10;  i++)   {
```

**Confidence... : 1**  *(high certainty)*
**Student:**  Makes the appropriate changes to the code.
**Resulting code:**

```
for ( int i = 1; i <= 10;  i++) {
      sum = smu + i;
}
```

**JITS:**  Would you like me to replace "smu" with "sum"?
*(Identifier replacement hint)*
**Student:**  Clicks Yes, or changes the code manually.
**Resulting code:**

```
for ( int i = 1; i <= 10;  i++) {
      sum = sum + i;
}
```

The tutoring process is dynamic. At any time the student is able to interject, disagree with JITS' suggestions, and modify the source code. JECA is designed to be invoked many times to support the JITS tutoring process.

JECA is significantly different from other standard Java compilers. Given the source program in figure 13, an ordinary java compiler would produce the following:

```
Test.java:5: ')' expected
      Forr (Int i=0; i <=10;   i++
                 ^
Test.java:5: not a statement
      Forr (Int i=0; i <=10;   i++
                       ^
Test.java:5: ';' expected
      Forr (Int i=0; i <=10;   i++
                                  ^
3 errors
```

Clearly the embedded JECA system in JITS is much more clear and helpful than standard Java compilers. JECA has been designed for the beginner Java programmer and intelligently recognizes the intent behind the student's code submissions.

## 6. Conclusions

JECA demonstrates a Proof of Concept that can be effectively used to assist beginner Java programmers. JECA was originally implemented in JFlex, CUP and JavaCC. However, it became clear that JavaCC offers the greatest control and flexibility over error recovery and error correction; therefore, future versions of JECA will be based on JavaCC.

JECA is a practical algorithm compiler that error corrects by intelligently learning and changing source program code, and identifies errors more clearly than current-day compilers. The goals achieved by JECA include:

i) intelligently recognize the 'intent' of the student;
ii) analyzing the student's code submission;
iii) 'auto-correcting', where appropriate (e.g., converting "While" into the keyword "while", "forr" into "for", etc.);
iv) learning individual student's misconceptions, and categorizes the types of errors he/she make;
v) producing a 'modified code' that will compile (or bring the code closer to a state of successful compilation),
vi) producing a 'modified code' that will meet program specifications (or bring the code closer to meet program specifications); and
vii) prompt the student programmer for information when necessary via well-defined hint support structures.

The ultimate goal of JECA is to give clear and helpful feedback to the student. In this paper, a Proof of Concept (i.e., JECA), was developed that fulfils the intended goals and assists the student to be able to learn programming better in a more enjoyably way in the Java Intelligent Tutoring System.

## 7. References

[1] G. Klein, *JFlex User Manual*, 2003. Retrieved September 3, 2003, from http://www.jflex.de/

[2] M.G. Burke & G.A. Fisher, A practical method for LR and LI syntactic error diagnosis and recovery, *ACM Transactions on Programming Languages and Systems*, *9* (2), 1987, 164-197.

[3] C. Fischer, R.J. LeBlanc, *Crafting a compiler with C.* (Redwood City, CA: Benjamin Cummings Publishing, 1991).

[4] A. V. Aho & T.G. Peterson, A minimum distance error-correction parser for context-free languages, *SIAM Journal of Computing, 1*, 1972, 305-312.

[5] E.R. Sykes, & F. Franek, (2003). A prototype for an intelligent tutoring system for students learning to program in Java, *Proceedings of the IASTED International Conference on Computers and Advanced Technology in Education*, Rhodes, Greece, 2003, 78-83.

[6] E.R. Sykes, Java$^{TM}$ intelligent tutoring system model and architecture, *Proceedings of American Association of Artificial Intelligence Spring Symposium on Human Interaction with Autonomous Systems in Complex Environments*, Menlo Park, CA, 2003, 187-193.

[7] J.R. Anderson, A. Corbett, K.R. Koedinger, & R. Pelletier, Cognitive tutors: lessons learned. *The Journal of the Learning Sciences, 4,* 1995, 167-207.

[8] K.R. Koedinger, Cognitive tutors, in K. D. Forbus & P. J. Feltovich (Eds.), *Smart machines in education* (Cambridge, MA: MIT Press, 2001) 145-167.

[9] A.C. Graesser, N.K. Person, Teaching tactics and dialog in autotutor, *International Journal of Artificial Intelligence in Education, 12*, 2001, 12-23.

[10] R.C. O'Reilly, & Y. Munakata, *Computational explorations in cognitive neuroscience* (London, England: MIT Press, 2000).

[11] A. Tucker, & R. Noonan, *Programming languages: principles and paradigms* (New York: McGraw-Hill, 2002).

[12] M. Pawlan, J2EE Tutorial, 2003. Retrieved December 30, 2003, from http://java.sun.com/j2ee/1.3/docs/