Automated debugging of Java syntax errors through
diagnosis

Ngozi V. Uti and Dr. Richard Fox
Department of Mathematics and Computer Science
Northern Kentucky University
Highland Heights, KY 41099

Abstract
Syntax errors arise in a program because the program code misuses the programming language. When these errors are detected by a compiler, they are not necessarily easy to debug because the compiler error message that reports the error is often misleading. JaSD, a knowledge-based system written in Java, is a program that can determine the cause of a given syntax error and suggest how to correct that error. JaSD uses artificial intelligence techniques to perform debugging. Currently, JaSD has enough knowledge to debug one of four different Java syntax errors. This paper describes the JaSD system and the diagnostic approach taken and offers some examples. Conclusions describe how JaSD can be expanded to debug additional errors.

Introduction
Syntax errors can arise in a program because the program code misuses the programming language. The compiler, in its attempt to translate the high-level language program into machine language, has reached an instruction that it does not understand. The error might be because the programmer misused the high-level language (e.g., improper punctuation or not structuring a statement correctly), but it is just as likely that the error is a careless omission or misspelling. The compiler detects syntax errors, but even so, syntax errors are not necessarily easy to debug. The debugging process requires identifying the code that caused the syntax error to arise and rewriting the code appropriately. The compiler gives the programmer assistance, but the process is not straightforward. Compiler errors may even be misleading.

The process of identifying the error's true cause and fixing it is a form of diagnosis or, more generally, credit assignment. Programmers are able to diagnose syntax errors with greater ease as they learn the language and gain more experience in debugging the language. This experience often comes in the form of understanding just how a given syntax error will arise. Surprisingly, a given error will arise because of a number of different situations, although the number of situations is usually fairly small. As an example, a common error in a language such as Java, C, C++, or Pascal is "semicolon expected." The semicolon-expected error is common because a semicolon is used in Java, C, and C++ to end instructions and in Pascal to delineate (separate) instructions. Forgetting a semicolon is common and yields the semicolon-expected error in any of these languages. Thus, the programmer learns how the error can arise. However, there are numerous possible causes for this error, omitting a semicolon being only one of them.

With the proper knowledge of how errors arise, a knowledge-based system can be implemented to perform diagnoses. Knowledge-based systems, sometimes called expert systems, are products of artificial intelligence research. As such, the intent of a knowledge-based system is to apply knowledge in a problem-solving setting to solve the problem that ordinarily would require a human, possibly a human expert. Knowledge-based systems, are particularly applicable within a restricted domain such as programming and syntactic debugging. This paper presents just such an approach in the construction of JaSD, a program that can determine the cause of, and in most cases generate a solution to a Java syntax error. Numerous approaches have been taken in attempting to automate the debugging process in computer programming. Both Allemang and Chandrasekaran (1991), and Hermenegildo et al (1999), take formal approaches by representing program code and seeking discrepancies against formal models. Cerecke (2001) and Fischer and Mauney (1992), however, attempt to repair specific syntax
errors when the compiler detects them by identifying code that could not properly be passed into grammatical syntax trees. For more information, Ducasse (1993) offers a survey of approaches.

JaSD, a program written in Java, solves a portion of the debugging problem by concentrating on four specific Java syntax errors. Specifically, JaSD approaches the debugging problem as one of diagnosis using classification. This research is a "proof of concept" to demonstrate that such a system could potentially debug all syntax errors of a given programming language using a classification-based artificial intelligence approach. Only time limitations forced the current system to be restricted to four syntax errors. Research has been done to capture knowledge needed to expand the system to handle several other Java syntax errors.

This paper introduces the idea of debugging syntax errors. It concentrates on Java syntactic debugging, offering the reader examples of how a syntax error can arise for different reasons. The paper then introduces the JaSD system, explaining how the system works. This is followed by examples of JaSD debugging each of the four syntax errors that have been implemented. Conclusions are offered to suggest how JaSD can be expanded.

Debugging Syntax Errors

Diagnosis is the process of credit assignment; that is, assigning the cause of symptoms or manifestations. In computer programming, debugging is a diagnostic process. Debugging occurs when an error arises; the programmer must then determine what instruction(s) might have caused the error and fix the instruction(s) to remove that error. One type of debugging is that of repairing syntax errors. A syntax error is an error caused because of a misuse of the programming language. The compiler detects syntax errors when the high-level language program is being translated into machine language. The compiler detects these errors because it is unable to translate a given instruction. The errors occur because the instruction did not make syntactic sense. When such an error is found, the compiler generates a syntax error message to the programmer.

One might think that fixing syntax errors is easy because the compiler tells the programmer what the error is (e.g., "semicolon expected") and where the error was found. However, the compiler error message is not necessarily an accurate description of the error. The semicolon-expected error arises when the compiler expects to find a semicolon but none is there. One situation occurs when the programmer has forgotten to include the semicolon. Unfortunately, many other causes are possible. In researching Java and this particular error, the authors have found the following four possible causes for this type of error:

- Incorrect usage of a semicolon (;)
- Incorrect usage of arithmetic operator (\(=, +, -, *, /, \) %)
- Incorrect usage of a comma (,) in a variable declaration
- Mishap with properly starting or ending a commented block of code

For causes 1, 2, and 3 above, the cause itself can be refined into more particular causes of a missing operator or a misplaced operator. There are approximately ten different sub-causes for the given error based on the above four general causes.

The compiler error message can be thought of as a manifestation of a malfunctioning program. The identification of why that error message was given is the process of syntactic debugging. By viewing debugging as a form of diagnosis, diagnostic solutions can be applied. The diagnostic process must then search through the possible causes of the given error. If these causes can be enumerated, then one need only consider each cause and determine if it is plausible. A test might be derived for this. For instance, if the instruction preceding the instruction in question is missing a semicolon, then a missing semicolon is likely. If the current instruction is an assignment statement that misuses an arithmetic operator, then incorrect usage of an arithmetic operator is plausible. Notice that in order to debug this error, the instruction identified by the compiler as containing the error may not actually be the instruction that caused the error. When a semicolon is expected, it is possible that the error was caused by the preceding instruction. In other syntax errors, the instruction that caused the error may occur many instructions prior to the instruction detected. So a search for the proper instruction must be performed. However, each error cause message (such as "incorrect
usage of the = operator") will communicate an expectation of where that instruction should be.

Therefore, the diagnostic process of debugging syntax errors consists of considering each possible error cause, searching the instruction(s) where that cause is expected to have occurred, and establishing whether or not the error is plausible through some test(s). To make the search process more efficient, if the error cause can have sub-causes, then the search continues by considering those sub-causes if the cause itself was found plausible. If that cause were found implausible, searching the sub-causes would be a waste of time. The diagnostic process is a classification-based approach of testing specific program instructions, using the tests that were previously used to identify if a given cause might be plausible. Once a plausible cause has been identified, fixing that error is straightforward as the solution is then identified to remove the cause. This classification-based approach is common in artificial intelligence (Chandrasekaran 1986, Bylander and Mittal 1986, Bylander et al, 1991).

Some examples follow in order to clarify the above discussion. One cause of the semicolon-expected error is the misuse of an arithmetic operator. In such a case, the Java compiler issued the "semicolon-expected" error because it thinks that the instruction has ended and that a semicolon is needed. Two such examples follow. The first is

$$a = b + c \cdot d;$$

Here, an arithmetic operator is missing between c and d. Java misinterprets this as being the statement

$$a = b + c;$$

where a semicolon was expected after c. In fact, the error is that an arithmetic operator (+, -, *, /) should have appeared between c and d. The instruction

$$a \cdot b + c + d;$$

is similar, except that the = operator was omitted. However, the same syntax error ("semicolon expected") is generated by the compiler. Notice that in both of these cases the error was not an omitted semicolon.

The solution to both of these errors is to insert the missing operator once the missing operator has been identified. In the first example, the operator must have been an arithmetic operator, although it is not possible to state with certainty which operator should have appeared. It is equally likely to have been +, -, *, /, or % (the modulo, or remainder, operator). In Java, an arithmetic expression cannot appear by itself but must appear inside another statement. In this case, the statement is an assignment statement; the instruction is assigning the value of b + c + d to the variable a. Therefore, the missing operator is the =, needed in all assignment statements. The solution to this error is simply to insert the = between the variables that lack an operator between them (a and b).

An Automated Java Syntactic Debugger

JaSD is an experimental program designed to debug Java programs. While numerous errors have been researched, JaSD has been implemented to debug four Java syntax errors. These errors include the following:

- **Semicolon Expected**
- **Right Parenthesis Expected (or ")" Expected**
- **Undefined Variable**
- **Else Without If**

These four errors were chosen to illustrate the range of complexity required in syntactic debugging. *Else Without If*, for example, can arise only if the reserved word *else* is found outside of an *If* statement; thus, it has few possible causes and the error itself must be located within the current block of code. *Undefined Variable* has few possible causes as well but might require searching the entire program to determine the correct cause. This is because variables can be declared anywhere in the program, and this error is commonly caused by a misspelled variable name. On the other side of possibilities, ")" Expected usually requires searching only the line of code where the error has arisen, but there are 12 possible sub-causes of this error that must be considered.
The research performed leading up to the implementation of JaSD was to identify all of the possible causes of a given syntax error. While approximately 15 syntax errors were examined, four were selected for JaSD. Each of these four syntax errors has its own list of possible causes, organized as mini-classification hierarchies. Each of these hierarchies has at least one level of possible causes, but there can be up to four levels of possible causes. Figure 1, for instance, illustrates the "(" Expected sub-hierarchy, which is the largest of the four hierarchies.

The "(" Expected error has three possible causes, two of which have sub-causes. The "(" Expected error will most commonly arise because of a problem with a parameter list or parenthetical expression. Parameter lists are used when an instruction calls another body of code, known in Java as a method. The specific cause of the "(" Expected error is a mishap with a ")", a mishap with some other punctuation operator (arithmetic operator, quotation mark or semicolon), or a misspelled variable that has spaces between the characters in the name.

A mishap with an operator can cause the compiler to believe that the parameter list or expression should have ended, whereas the programmer intended it to continue. For instance, if an arithmetic operator is omitted between two variables in a parameter list, the error will arise. This is the case with the following statement, which calls the square root method. Since square root expects a single parameter and two were found, the Java compiler concludes that the error is a missing ")" after the first parameter.

\[ x = \text{Math.sqrt}(a \ b) \]

In fact, the programmer meant to apply an arithmetic operator to the variables a and b first yet omitted it. As with a previous example, it is not possible to determine what arithmetic operator was intended, but the error is not a missing ")" but instead a missing arithmetic operator.

The "(" Expected error also arises if a String literal is passed as a parameter but does not have a beginning quote mark. An example of this is

\[ \text{System.out.print}(\text{Hello World}) ; \]

Another situation occurs when the programmer inserts a ",;" into the parameter list by accident. Here, the compiler confuses the ",;" used to indicate the end of the instruction with a part of a String parameter. Consider these two instructions:

\[ \text{System.out.println}(a; b) ; \]
System.out.println(a b);

In both of these instructions, the ")" Expected is provided by the compiler between the first and second parameters. In the first example, the programmer used a semicolon to separate the two parameters instead of a "+". In the second example, it is less clear whether the error arose because the programmer omitted a "+" between a and b, or if a b is a misspelled version of ab. If there has been a variable ab declared previously, then the latter cause is more likely. However, if the programmer has declared three variables, a, b, and ab, then either cause ("+" omitted or misspelled "ab") is equally likely.

In the case of a mishap with a ")", the error may actually have arisen because the ")" is missing or misplaced. An examination of the current line of code will determine which is the case. If a ")" is found, but not where it is expected, then it is likely that the programmer has simply placed the ")" in the wrong location.

The Semicolon Expected, ")" Expected, and Else Without If are fairly straightforward errors in that a specific cause can usually be determined without uncertainty. A fourth error, Undefined Variable, was researched because it cannot be resolved by merely considering the syntax of the program. Instead, to resolve this error, intention must be known and so an automated system, which merely considers syntax, can at best only enumerate possible causes and solutions. It would then be up to the programmer to select which candidate, as generated by the system, is the most likely cause of the Undefined Variable error.

There are many situations where a misspelled variable can cause an error. One example is shown above where ab is misspelled as a b and leads to a ")" Expected error. Usually, however, misspelling a variable results in an Undefined Variable error. The cause of an Undefined Variable error can be one of several possibilities: (1) a variable (or constant) is declared and later misspelled, (2) an unknown word is actually a misspelled Java reserved word, (3) an unknown word is a misspelled method or class name that has been imported from a library, or (4) the unknown word is actually a variable that the programmer forgot to declare. Determining which of these is the case requires several pieces of knowledge: (1) the reserved words of Java, (2) all variables and constants declared in the class, (3) all constants declared in imported classes, (4) all methods declared in imported classes.

To determine if an unknown word is actually a misspelled reserved word as opposed to a declared variable, one must compare the unknown word to the known entities and determine how closely it matches. Consider an unknown word "foo" in a program that has a declared variable "fog." Is "foo" a misspelling of "fog" or a misspelling of the reserved word "for"? Further, imagine a program that has declared the variables "number1," "number2," "number3" and has found an unknown word "number4." Is the error that "number4" was a misspelling of one of the declared variables, or an undeclared variable? Syntax may help determine the former situation by determining if the unknown word "foo" appears prior to what looks like a for-loop definition. The latter case cannot be determined without understanding the role that "number4" is playing in the program, which is beyond the scope of a strictly automated process and must involve a human.

To perform the comparison between the unknown word and all known entities (reserved words, declared variables, imported classes and methods), a string comparator is applied. This comparator matches the unknown word against all known words in the given Java program and ranks each known word in order of how closely it matches. The known words are all Java reserved words, all variables and constants declared to this point in the given Java program, and all class names and methods imported from Java libraries. Each word is matched, generating a "proximity" rating of how closely it matches. This proximity is based on whether or not the proper characters are found in approximately the proper locations. The word "foo" very closely matches "for" and "fog" but does not closely match "of" or "float". Closer matches are given for words with the same number of characters or if a character is in the proper place but upper-case instead of lower-case or vice versa. If a character is found but out of place, the match is not as close. Finally, if a character is expected but not found, the match is even less close. It should be noted that this comparison operation is used in other errors such as ")" Expected, but is used extensively in the Undefined Variable error.

The research into JaSD has been more of a "proof-of-concept" to demonstrate that syntactic debugging can be done in an automated setting using Artificial
Intelligence techniques. JaSD itself is envisioned as a tool for introductory Java programmers so that the introductory Java programmer can be assisted in syntactic debugging. It identifies the cause of a compiler error message so that, in turn, the programmer can learn more about that particular syntax error, thereby potentially making the programmer a better debugger. Further, JaSD could possibly be incorporated into a compiler so that the compiler can not only detect but also fix syntax errors automatically, thus alleviating this task from the programmer. However, that was not the intention of this work. JaSD is the implementation of the four syntax errors, the String comparator, and all other necessary code. The four syntax errors are implemented as a series of tests that determine the plausibility of the given error sub-cause, and if found plausible, tests to determine the plausibility of each sub-cause's sub-cause. Finally, JaSD contains the ability to generate a message to explain how to fix the given error once a specific sub-cause has been identified as plausible. In the case of a misspelled entity, numerous messages might be generated because JaSD is not able to identify which entity is misspelled.

JaSD is far from complete as there are over 100 possible syntax errors in Java. Adding more syntax errors to the system is anticipated as merely being a case of identifying the causes of the syntax error and building the hierarchy and any matching facilities needed for those causes. The comparator operation can be used by many other errors to determine if the cause of some error is in fact a misspelled reserved word, class, method, constant, or variable, simplifying any added error capabilities to JaSD.

JaSD does place three restrictions on any Java program being debugged. First, the source program cannot use the multi-line form of comments embedded between /* */ symbols, although these symbols can be used, but only if the comment is contained within a single line. Second, any imported classes must be imported by name rather than by using the * character. For example, “import java.io.BufferedReader;” is acceptable but “import java.io.*;” is not. Third, some situations require white space placed between punctuation and reserved words; for example, “for(j=0; j<n; j++)” would have to be written as “for (j=0; j<n; j++)”. Java does not require the white space, but JaSD does. All three of these restrictions are placed but simplify the implementation of JaSD, and none are limitations of the diagnostic approach. Future work might include the removal of these three simplifying restrictions.

Examples

Three examples are presented in this section to demonstrate JaSD in action. The first two examples contain code that cause the "")" Expected syntax error but have causes other than a missing or misplaced ")". The third demonstrates the String comparator for the Undefined Variable syntax error. The first example is shown in Figure 2. Here, the compiler has presented the programmer with a ")" Expected error on line 13. The instruction that caused the error is:

System.out.println("This is a test" + value); // Prints value

The compiler generates the error because a semicolon (;) is reached before a right parenthesis (")"), confusing the compiler. The error generated is that a ")" was expected, and a programmer might decide that the solution is to place a ")" at that location. While this will fix the "") Expected error; it will cause at least two new errors to be generated:

not a statement

";" Expected

So it is much better for the programmer to learn why the error arises rather than simply trying to fix it.

JaSD solves the problem by first considering if the error has arisen because of a misspelled variable name. Since value is the only variable in the instruction, and it is properly spelled compared to its declaration, this is not the case.

Next, an operator problem is considered. In this case, an operator is found in the midst of the parameter list, so this is possible. Refining the cause, JaSD identifies the operator as ";", which is one of the operators that can cause this syntax error. It confirms this is the
cause by checking to see if the ";" appeared at the end of the statement or not. Since ";" appears before the end of the statement, this is a likely cause of the error. To be complete, JaSD examines the last possible cause, a mishap with ")". Since a ")" does appear in its proper location, this cause is ruled out. Now that JaSD has identified the cause, its solution is to remove the inserted ";" from the location in which it was found.

The second example, presented in Figure 3, is like the first example in that the compiler has generated a ")" Expected error. The process is also similar to the first example. First, JaSD attempts to find a misspelled variable in the line. The only variable found has been properly spelled. Next, it looks for a problem with an operator. In this case, it finds an open and closed quotation mark. However, it also finds two different items in the parameter list without a separator. So, in this case, it checks to see if there is some arithmetic operator in the line. There is none, and so it identifies the cause of the error as a missing arithmetic operator as opposed to a misplaced arithmetic operator. This is further refined into missing "+" because the "println" statement expects String arguments and the only String operator is the "+" sign. The final cause, a ")" problem, is ruled out because the ")" is found exactly where it should be. JaSD responds with the message "Missing + sign before 'strValue' Pos: 43" both stating the problem and where the "+" sign should be inserted to fix the problem.

The final example, shown in Figure 4, illustrates the use of the String comparator where the Undefined Variable syntax error has arisen because the item "values" has not been declared. JaSD creates a list of all reserved words, declared variables, and constants. There are no imported classes, so there is no need to examine class methods or constants. Next, the String comparator compares each item in the list to "values" to find the closest match. In this case, only the variable "value" is a close match. None of the reserved words come close to matching, and none of the other variables match as closely as "value." Interestingly, the variable "Kelly" has the next highest proximity rating, whereas "strValue" has a much lower proximity rating than might be expected from its similarity to "values." However, JaSD does not rule out these other matches, and so all four are presented to the programmer as possible causes of the syntax error. This error may be caused because "values" is
(1) misspelled in the line where the error arose, (2) correctly spelled, but "value" was misspelled when declared, (3) correctly spelled, but the programmer failed to declare it, and (4) correctly spelled, but one of the other variables was misspelled when declared.

It is impossible for JaSD to identify which of these is the correct cause. So, the result is a list of possibilities for the programmer to consider.

Conclusion

This paper has demonstrated the JaSD system. JaSD, the Java Syntax Debugger, uses Artificial Intelligence techniques to find the cause of a given Java syntax error. The purpose of this system is not necessarily to automate the debugging task, although that is certainly a possibility. Instead, this paper has attempted to demonstrate a proof of concept that syntactic debugging can be automated in a fairly straightforward manner. To that end, JaSD has been tested on numerous Java programs that suffer from the four selected errors. As expected, JaSD is able to identify correctly the cause of the error when the cause is identifiable as strictly syntactic, and also, if the cause is a misspelled entity that JaSD cannot identify, then JaSD is able to display properly the list of known words that may be misspelled.

While the current system debugs only 4 of the over 100 possible syntax errors in Java, the approach taken is one that would easily permit JaSD to be increased in scope. The reason for this is that each of the syntax errors uses its own knowledge to identify that error. This knowledge comes in the form of a hierarchy of error causes and sub-causes, and tests that are used to determine the plausibility of each of those causes/sub-causes. To expand the system to include identification of other errors, one need only identify that error's causes and implement the tests for each of those causes. Some of the other syntax errors considered for implementation and future work include:

- Illegal Start of Expression,
- Cannot Resolve Symbol,
- Not a Statement,
- While Expected, and
- ")" Expected.
Figure 4. Undefined Variable Example

References


New Natural Science building, dedicated August 2002