JAVA ERROR CORRECTION ALGORITHM

A JAVA ERROR CORRECTION ALGORITHM IN THE FRAMEWORK OF AN INTELLIGENT TUTORING SYSTEM

By

EDWARD R. SYKES, M.Ed., B.Ed., B.Sc.

A Thesis

Submitted to the School of Graduate Studies

in Partial Fulfillment of the Requirements

for the degree

Master of Science

McMaster University

MASTER OF SCIENCE (2005) McMaster University

(Computer Science) Hamilton, Ontario

TITLE: A Java Error Correction Algorithm in the Framework of an Intelligent Tutoring System

AUTHOR: Edward R. Sykes, M.Ed. (Brock University), B.Ed. (The University of Western Ontario), B.Sc. (McMaster University)

SUPERVISOR: Professor Frantisek Franek

NUMBER OF PAGES: vi, 91

Sections of this thesis have been previously published in the IASTED International Conference on Advances in Computer Science and Technology conference proceedings:

Sykes, E. R., & Franek, F. (2004). Presenting JECA: A Java Error Correcting Algorithm for the Java Intelligent Tutoring System, *Proceedings of the IASTED International Conference on Advances in Computer Science and Technology*, St. Thomas, Virgin Islands, USA (pp. 151-156).

Abstract

Intelligent Tutoring Systems (ITS) are, in many respects, very similar to human tutors. Based on cognitive science and Artificial Intelligence (AI), ITS have proven their significance in many disciplines. Currently, ITS can be found in core Mathematics, Physics, and Language courses in hundreds of schools across Canada, the United States, and various countries in Europe. ITS are growing in acceptance and popularity for reasons including: i) increased student performance, ii) deepened cognitive development, and iii) reduced time for student to acquire skills and knowledge.

Bloom (1984) showed that one-on-one human tutors could increase the average student's performance to the ninety-eighth percentile in a standard classroom. Furthermore, in order for students to reach their potential, individualized tutoring is a necessity. Intelligent Tutoring Systems have demonstrated that student achievement is 1.0 standard deviation higher than typical classroom environments. Another benefit ITS have is the speed of knowledge acquisition. Students learning from an ITS have completed problems in one-third of the time compared to students in the control group.

This thesis focuses on the design and implementation of an error correction algorithm in the specific context for use in an ITS for the Java programming language. The Java Error Correction Algorithm (JECA) was designed to be used by first year College and University students with little or no programming experience. JECA attempts to determine the "intent" of the student's submission by rigorously analyzing the student's code. Behind the scenes, JECA makes changes to the student's submission in order to facilitate this analysis. However, once JECA determines the most reasonable intent of the student, these changes are made known to the student. The results from JECA are passed to the Java Intelligent Tutoring System (JITS) in the form of hints and suggestions, which are then used for instructional purposes.

This thesis focuses on JECA, however, to ensure contextual relevance and significance, the Java Intelligent Tutoring System is included. JITS is implemented using advanced e-learning technologies and its multi-threaded distributed architecture makes JITS scalable, robust and easy to maintain. JITS supports personalized student development by tracking and modeling every student in the system. JITS is an online website always available for students and requires only a browser and an internet connection.

Table of Contents

List of Tables

Page

List of Figures

CHAPTER ONE: THE PROBLEM

Context

Java has grown into a very popular programming language. In fact, it is one of the most popular programming languages in the world for internet applications (Chen, 2004). At numerous educational institutions, Java is now the core language used in Computer Science and Software Engineering programs. For instance, at the Sheridan Institute of Technology and Advanced Learning over 3000 students now use Java. I have been a member of the School of Applied Computing and Engineering Sciences at Sheridan for over a decade and have gained a lot of experience in teaching programming to students from first year to final year to graduate level students. During the last several years, I became interested in Intelligent Tutoring Systems (ITS) and began to conduct research. I discovered a number of key issues that give rise to this thesis research. There currently is no ITS for the Java programming language. Secondly, there is no ITS that analyses the student's submitted code in such rigor as the system developed in this thesis. Third, no other ITS actually modifies the student's submission in numerous ways in an attempt to determine the intent of the student. Fourth, the system developed is a multi-threaded distributed internet-based application that is always available and to use requires only a simple browser and an internet connection.

The net result of this research is intended to be extremely practical. The system is being field-tested by first year programming students at the Sheridan Institute of Technology and Advanced Learning. It is my hope that many educational institutions will use the system including high-schools, Colleges and Universities.

Introduction

The system proposed in this thesis is entitled the "Java Error Correction Algorithm" (JECA) for the Java Intelligent Tutoring System. The JECA module is a major component of this thesis for various reasons. JECA "reasons" over the submitted code and makes intelligent changes to the code in an attempt to bring the submitted code closer to a state of successful compilation. As a result, a great deal of effort was placed on the topic of compilers and error recovery strategies. Error recovery in compiler design and construction has traditionally been focused on analyzing the input source code and identifying as many errors as possible in as short a period of time (Aho, Sethi, & Ullman, 1988; Fischer & LeBlanc, 1991). Most compilers of today do not afford the luxury of sophisticated error diagnosis or recovery. There are many reasons that justify this approach. One, the speed of compilation is very important and the user is now more than ever unwilling to wait patiently for the output of compilation. Second, complex error correction strategies take time and are generally not efficient for medium to large size programs. Third, the compiler's output is designed to be used by a professional programmer who understands programming, and knows how to interpret the compiler's output without difficulty. These reasons and others push the designers and maintainers of compilers in different directions other than error correction.

However, there are certain circumstances where a sophisticated error correction strategy is very fitting and beneficial to the target audience. For beginner programmers an "intelligent" error correction strategy is very helpful and advantageous. Typically, the role of the compiler, when errors are encountered, is to identify all possible errors in one

pass of the source code (Aho et al., 1988; Fischer & LeBlanc, 1991). The compiler generates error messages that are often quite cryptic making them difficult to understand and troubleshoot for the beginner programmer. Additionally, cascading error messages and false errors may also be generated. These factors gave the motivation and justification for the design and development of the systems in this thesis.

The Java Error Correction Algorithm presented in this thesis was designed for the Java Intelligent Tutoring System to be used by beginner Java programmers. Since the size of the code that will be analyzed by the system will be very small, the speed of compilation is not an issue. As a result, additional processing and analysis may take place to better understand the intent of the beginner programmer and to offer clear and meaningful feedback to the programmer.

Problem Statement

Today's compilers perform error recovery but 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 recovery 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.

In contrast to today's compilers, in certain circumstances, as in learning to program, it is more desirable to have the compiler act "intelligently" and make "intelligent" changes to the source program. This thesis research examined error correction in a specific context involving small Java programs. A review of various tools is presented including JFlex, CUP, and JavaCC (Hudson, 1999; Klein, 2004; Norvell, 2004; Sreenivasa, 2004).

This thesis presents JECA and JITS. JECA is a practical algorithm for a compiler that error corrects by intelligently changing code and identifies errors more clearly than other error recovery/correction systems. The goals of the proposed system are to:

i) analyze the student's code submission;

ii) intelligently recognize the "intent" of the student;

iii) "auto-correct" where appropriate (e.g., converting "forr" into "for", etc.);

iv) learn individual student's misconceptions, and categorizes the types of errors s/he makes;

v) produce a "modified code" that will compile (or bring the code closer to a state of successful compilation); and

vi) 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 messages so that the Java Intelligent Tutoring System may provide valuable feedback for the student similar to human-tutoring dialogue sessions. In this way, the student will be able to learn

programming better and more enjoyably. The following is the underlying research question addressed in this thesis: "Is the proposed solution involving JECA a suitable approach for intent recognition for the Java Intelligent Tutoring System?"

Rationale

Traditionally, it was extremely beneficial to the programmer for the compiler to carefully go through all of the source code until the end and ascertain the 'correct' meaning of the code (Aho et al., 1988; Bennett, 1996; Fischer & LeBlanc, 1991). As a result, the compiler often produced many erroneous messages (Aho et al., 1988; Bennett, 1996). Unfortunately, times have not changed. Compilers of today still produce the same type of output and attempt to recover from errors in much the same way as compilers of the past.

By changing code and identifying errors more clearly than other current-day compilers, JECA will be beneficial for the target audience – beginner programmers (Aleven & Ashley, 1997; Sykes, 2003). The first part of this thesis presents the Java Error Correction Algorithm – a practical algorithm for small Java programs to be error corrected in an intelligent way. JECA is different from standard compilers in that JECA:

i) produces one main error message;

ii) stops parsing if the current production cannot be parsed;

iii) encourages the student to address the main problem and to correct it;

iv) reduces student anxiety from getting overwhelmed with numerous error messages; and

v) focuses the student on the problem at hand.

The goal of this research is to determine if JECA is a suitable core component for the intelligence behind the Java Intelligent Tutoring System. The second part of this thesis is the design and development of JITS.

Outline of Remainder of Document

Chapter Two focuses on reviewing appropriate tools used in the research and development of JECA and JITS. The tools investigated for JECA were: JFlex with CUP and JavaCC. An overview of lexical analysis and parsing is presented. Additionally, the specifics of JFlex, CUP, and JavaCC are discussed in relation to error recovery capabilities, and potential to implement specific error recovery algorithms. The last section of this chapter presents a summary of the current state of Intelligent Tutoring System research.

Chapter Three describes the design of the Java Error Correction Algorithm. The scanner and parser components of JECA are presented in detail. The last section of this chapter describes the design layout for the Java Intelligent Tutoring System with JECA being the core module.

Chapter Four presents the implementation of JECA and the Java Intelligent Tutoring System. Human-Computer Interaction mechanisms, Hint Generation and User Modeling techniques are discussed in this chapter.

The last chapter of this document summarizes the results and discusses the implications of the analysis. In this chapter the proposed research question is addressed. That is, "Is the proposed system involving JECA a suitable approach for intent recognition for the Java Intelligent Tutoring System to be effective?" The recommendations section of this chapter offers a critical review of JECA and the Java Intelligent Tutoring System and provides direction for future research.

CHAPTER TWO: LITERATURE REVIEW

This chapter presents a review of appropriate tools and current state of research in the area of error recovery, error correction and Intelligent Tutoring Systems. This literature review was important in order to design and develop JECA. The tools investigated for the design and construction of JECA were: JFlex, CUP and JavaCC. These tools were selected for analysis because they are the equivalent to the standard LEX and YACC tools traditionally used in compiler research. JFlex, CUP and JavaCC were significant tools in this research because they generate Java compliant code and offer full support for Java code integration. An overview of lexical analysis, and parsing is discussed in this Chapter. Additionally, the specifics of JFlex, CUP and JavaCC are presented in relation to error recovery and correction capabilities and the potential each of these tools offer to implement specific error recovery algorithms.

A thorough literature review was conducted on the topic of error correction in compilers. At the time of this thesis, there were no publications available in this specific area of Computer Science for the Java programming language. However, there were theoretical designs and implementation in other languages which proved to be helpful for determining how to best conduct this thesis' research. The comparison between the proposed Java Error Correction Algorithm and other error correction designs is quite technical and relies heavily on implementation details. As a result, these details may be found in Chapter 4 – the implementation chapter of this thesis.

The last section of this chapter summarizes the current state of research in the field of Intelligent Tutoring Systems since JECA is intended to be 'plugged-in' to an ITS.

Lexical Analysis Overview

Lexical analysis is the first phase in the process of compilation. The main task for the lexical analyzer, often referred to as a lexer, or scanner, is to read the input characters and produce as output a sequence of tokens that the parser uses for syntax analysis. A token is a segment of text, regardless whether it be readable or comprised of symbols. Tokens are generally defined abstractly in a context free grammar, which is fed into a program such as JavaCC which checks the stream of tokens for conformity to this grammar. In order for the parser and scanner to understand the tokens a symbol table is used. Figure 1 depicts a typical schema for the lexical analyzer. With reference to Figure 1, the arrows indicate the flow of data and/or method invocation. For instance, the Parser issues the message 'getToken' to the Lexical Analyzer which in turn sends a token to be consumed by the Parser. The double arrow line between the Symbol table and the Parser indicates that the Symbol table is updated during the parse operation but it is also referenced by the Parser during the operation.

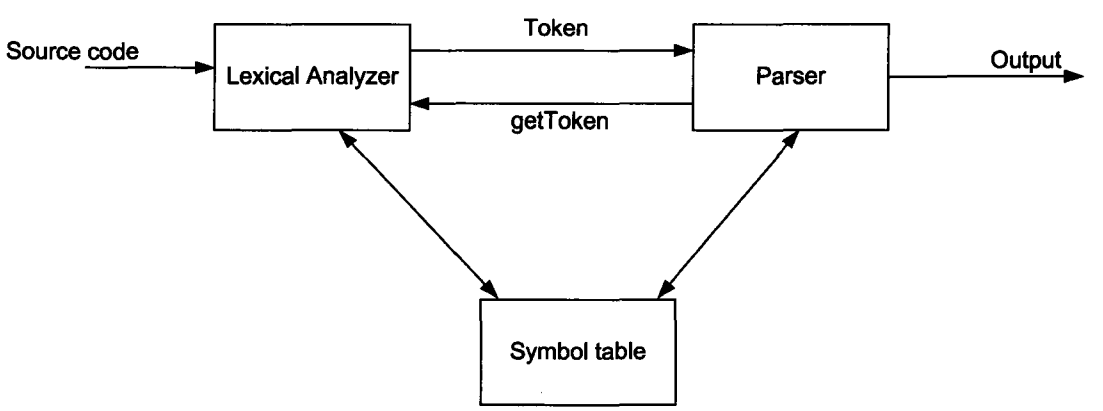


Figure 1. Lexical Analyzer and Parser communication.

Given the following simple method in the Java programming language:

```
public String getName() {
   return _name;
}
```

the Lexical Analyzer would perform a series of actions to produce tokens for the parser. Figure 2 depicts another view of the Lexical Analyzer using terminology of the developer, users and maintainers of JavaCC.

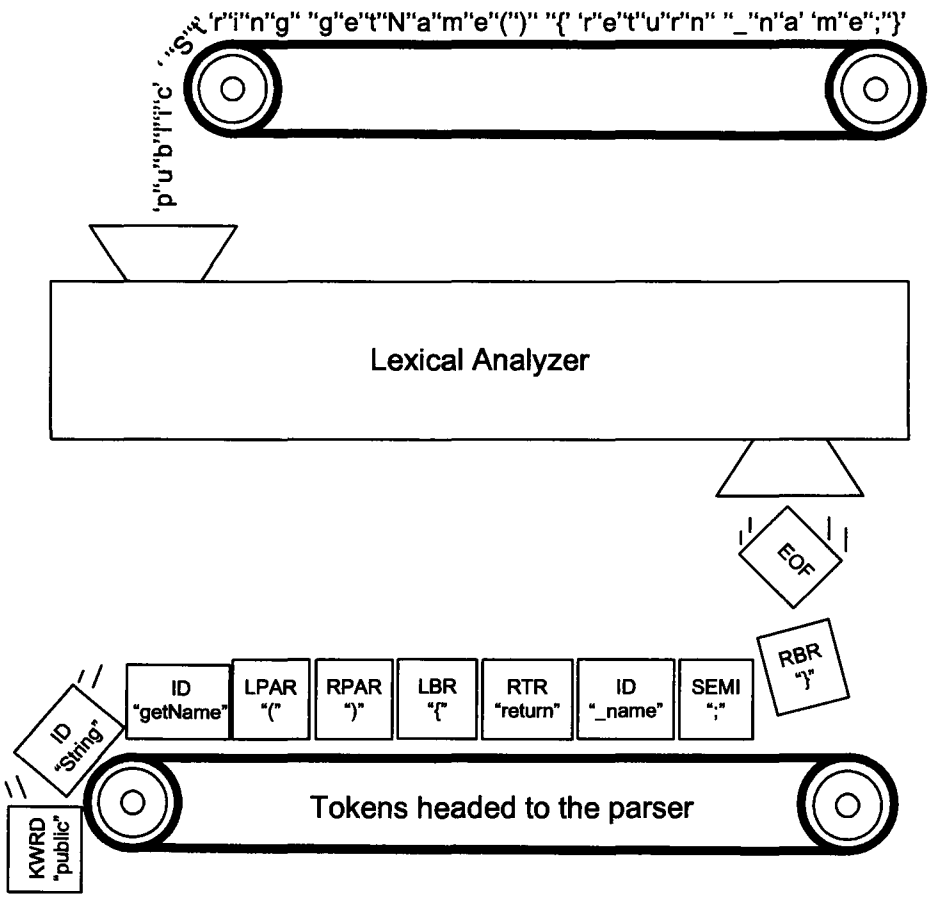


Figure 2. Token Manager consuming the input stream and producing tokens.

Parsing Overview

The parser's responsibility is to consume the sequence of tokens, analyze its structure, and produce something via a generator. The 'something' that the generator produces is up to the developer. It could be three-address code, an XML document, an abstract syntax tree, or many other forms of output. Figure 3 describes the relationship between the token and the parser.

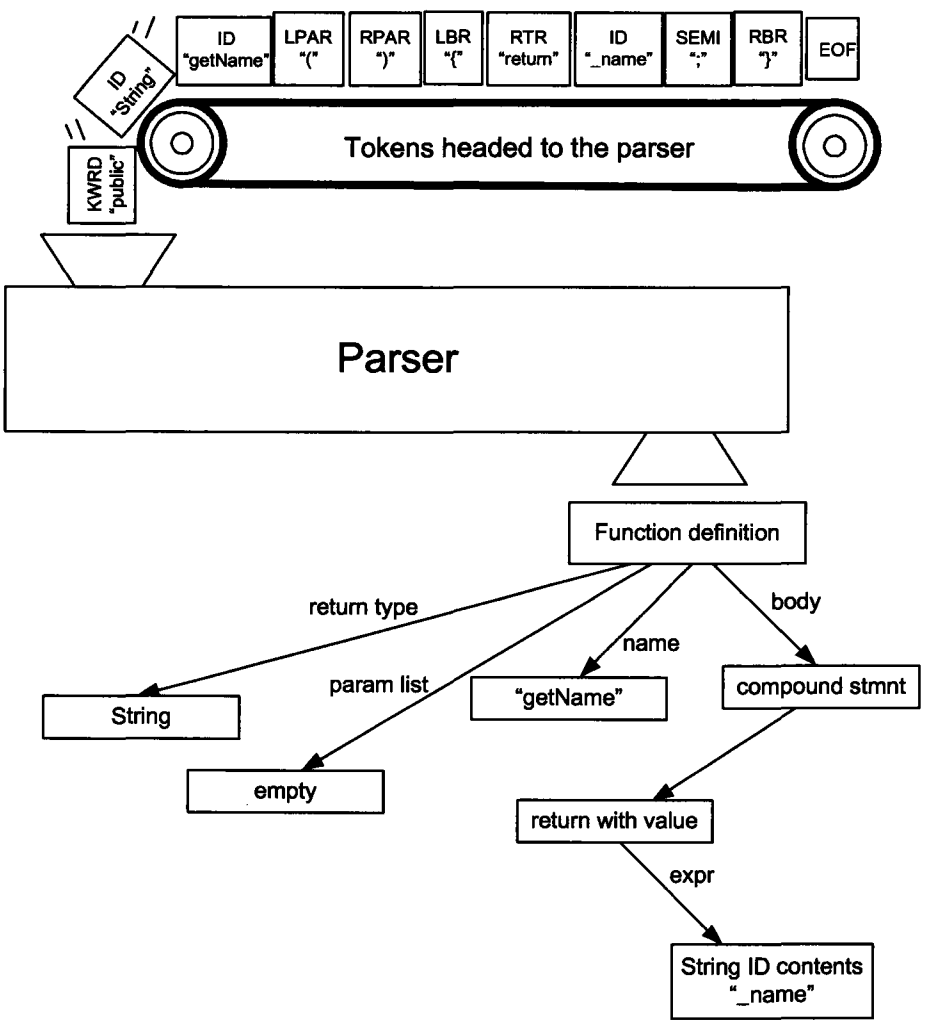


Figure 3. The Parser analyzes the sequence of Tokens received from the lexical analyzer.

A review of parsing is necessary to appreciate the complexities involved in addressing the problem of error correction. The following list includes the types of parsers and the corresponding grammars they can accommodate.

Recursive Descent: A recursive descent parser is a top-down parser built from a set of mutually-recursive procedures or a non-recursive equivalent where each such method usually implements one of the production rules of the grammar. The structure of the resulting program closely mirrors that of the grammar it recognises. For example, consider the following grammar in Backus Naur Form (BNF):

<expr_1> ::= NOT | () | TRUE | FALSE
<expr_2> ::= AND | OR

The following functions parse the input stream.

```
function parse_expr_1() {
   if inp = 'N' then ( read('N'); read('O'); read('T'); parse_expr_2() );
   if inp = '(' then ( read('('); parse_expr_1(); parse_expr_2(); read(')') );
   if inp = 'T' then ( read('T'); read('R'); read('U'); read('E') );
   if inp = 'F' then ( read('F'); read('A'); read('L'); read('S'); read('E') );
}

function parse_expr_2() {
   if inp = 'A' then ( read('A'); read('N'); read('D'); parse_expr_1() );
   if inp = 'O' then ( read('O'); read('R'); parse_expr_1() );
}
```

These procedures use a global variable *inp* that contains the current character in the input stream. The procedure `read(inp)` reads in a character from the input stream.

LL Parsers: An LL parser is a table-based top-down parser for a subset of the context-free grammars. It parses the input from Left to right, and constructs a Leftmost derivation of the sentence. The class of grammars which may be parsed in this fashion is known as the *LL grammars*.

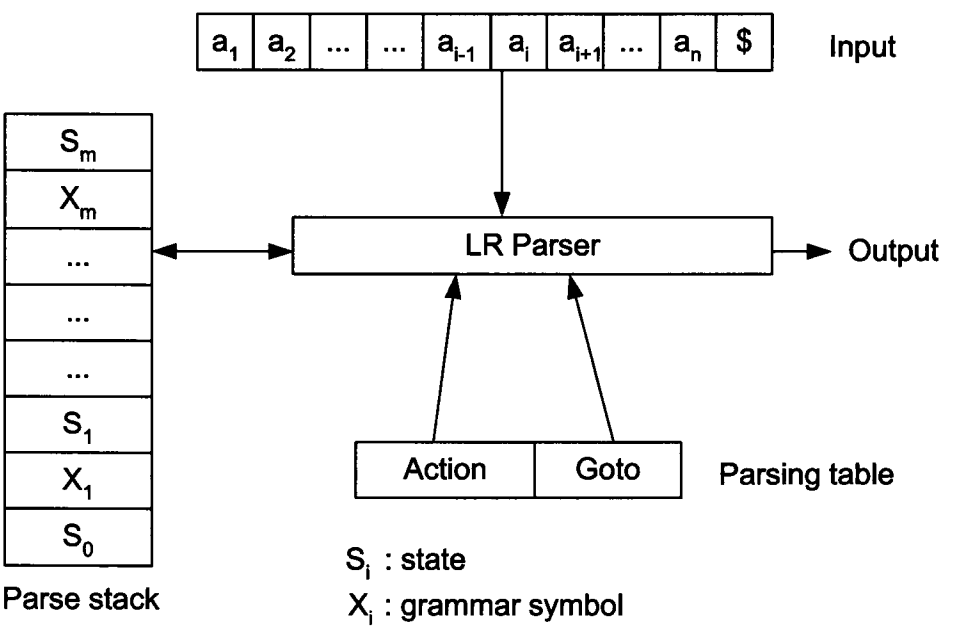
LL(k): LL(*k*) parsers uses *k* tokens for look-ahead when parsing the input stream. LL(1) grammars, although fairly restrictive, are very popular because they only need to look at the next token to make their parsing decisions. LL(k) parsers, on the other hand, must be able to recognize the use of a production after seeing the first k symbols of its right hand side. LL(k) parsers where k>1 are rare because of the additional complexities involved in the implementation of such parsers (Aho et al., 1988).

LR(k): These types of parsers are sometimes referred to as "bottom-up parsers". Unlike LL(k) parsers, LR(k) parsers are implemented using a bottom-up design. LR parsers read their input from **L**eft to right and produce a **R**ightmost derivation; *k* refers to the number of unconsumed "look ahead" input symbols that are used in making parsing decisions. An LR parser has an *input buffer*, a *stack* on which it keeps a list of states it has been in, an *action table,* and a *goto table.* The *goto table* indicates what new state the parser should move or which grammar rule it should use given the state it is currently in and the terminal or nonterminal it has just read from the input stream. All bottom up parsers have a similar algorithm:

```
loop
  1) try to find the leftmost node of the parse tree which has not been
     constructed (but all of its children have been constructed - the
     sequence of children is called the handle)
  2) construct a new parse tree node (this is called reducing)
end loop
```

The parsing process in bottom-up parsers can be thought of as "handle pruning". During parsing conflicts can arise if the grammar is written in a manner that two or more productions may match a specific kind of input. Left recursion is also supported by LR

parsers which can lead to conflicts as well. LR parsers use a "parse stack" which is a stack used in the parsing operation containing elements of symbols with a corresponding state. Reduction is the action of replacing the handle on the top of the parse stack with its corresponding left-hand side. Shifting is the action of moving the next token to the top of the parse stack. A model of an LR parser is shown below:



For an example, consider the following grammar

```
(1) E ::= E * N
(2) E ::= E + N
(3) E ::= N
(4) N ::= 1
(5) N ::= 2
```

Given an input stream of "2 + 2", the LR parser would use the action and goto tables to shift and reduce the input. At the end of the processing, the rule numbers that will be used are: [2, 5, 3, 5] which is a rightmost derivation of the string "2 + 2".

The following is another example of the operation of an LR(k) parser. Given the following grammar:

```
Sentence  ::= Subject Verb Object .
Subject   ::= I | a Noun | the Noun
Object    ::= me | a Noun | the Noun
Noun      ::= duck | bird | turtle
Verb      ::= like | is | see | sees
```

and the input stream: "`The duck sees a turtle .`", the parse tree grows from the bottom (i.e., leaves) up to the top (i.e., root). The input stream is read left to right and the right-derivations are read backwards as follows. First, the word "The" is matched but no production rule is executed yet. Next, the word "duck" is matched causing the production "Noun ::= duck" to be executed. The "Subject ::= The duck" production is executed next. The parser then receives the token "sees", which matches the production "Verb ::= sees". The parser next receives the "a" and then the "turtle" tokens. Upon reading the "turtle" the following production is fired: "Noun ::= turtle", which causes the higher level production: "Object ::= a turtle" to be executed. Lastly, the period token (i.e., ".") is read, resulting in the desired match for a sentence: "Sentence ::= Subject Verb Object ."

Despite the advantages of LR parsers being able to recognize a larger number of possible grammars than LL parsers, there are specific problems associated with LR algorithms. LR parsers can encounter shift-reduce and reduce-reduce conflicts. A shift-reduce conflict arises when the algorithm cannot decide between a shift action or a reduce action. A reduce-reduce conflict occurs when the algorithm cannot decide between two (or more) reductions (for different production rules).

LALR(k): A Look-ahead **LR** (left to right) parser is a specific type of LR parser. These types of parsers are very popular type because they give a good trade-off between the number of grammars they can deal with and the size of the parsing tables it requires. It is these types of parsers that are generated by compiler-compilers such as YACC and GNU Bison.

There are many advantages and disadvantages when considering the various types of mechanisms for parsing an input stream. Table 1 depicts these advantages and disadvantages.

Table 1 *Types of Parsers – Advantages and Disadvantages (sources: (Aho et al., 1988; Fischer & LeBlanc, 1991))*

Type	Advantages	Disadvantages
Top-down recursive decent (a form of LL(1))	- Fast - locality - simplicity - error detection	- often hand-coded - maintenance - no left-recursion
LL(1)	- simple method - fast - automatable - error detection	- LL(1) is a subset of LR(1) - no left-recursion
LR(1)	- fast - automatable	- table size - error recovery
LALR(1)	- large number of grammars - fast - automatable	- table size - error recovery

CUP versus JavaCC

The Java Based Constructor of Useful Parsers (CUP) is a system for generating LALR parsers from simple specifications. It serves the same role as the widely used program YACC. CUP offers most of the features of YACC (Hudson, 1999). However, CUP is written in Java, uses semantic actions including embedded Java code, and produces parsers that are implemented in Java.

JavaCC is a Java parser generator written in the Java programming language (Norvell, 2004). The community of JavaCC users have developed a collection of grammars including Java 1.0 through to the current version 1.5 specifications (Sreenivasa, 2004). It is similar to CUP but has the following distinct features:

Top-Down: JavaCC generates top-down (recursive descent) parsers as opposed to bottom-up LALR parsers generated by YACC, CUP and similar tools. Top-down parsers have a numerous advantages such as being easier to debug, having the ability to parse to any non-terminal in the grammar and having the ability to pass values up, as well as, down the parse tree during parsing.

Large User Community: JavaCC is by far the most popular parser generator used with Java applications. There are many thousands of participants using JavaCC.

Lexical and Grammar Specifications: The lexical specifications such as regular expressions, strings, and the grammar specifications are both written together in the same file. This is quite different from JFlex and CUP where there is a distinct separation of responsibilities for the scanner and parser. In JavaCC there is only one file for the scanner and parser which makes grammars easier to read. Furthermore, since it is

possible to use regular expressions inline in the grammar specifications in the form of attributed productions, it makes it easier to maintain.

Syntactic and Semantic Lookahead Specifications: By default, JavaCC generates an LL(1) parser. However, there may be portions of the grammar that are not LL(1). JavaCC offers the capabilities of syntactic and semantic lookahead to resolve shift-shift ambiguities locally at these points. For example, the parser is LL(k) only at such points, but remains LL(1) everywhere else for better performance.

Consequently, LL(1) parsers (including top-down parsers) do not encounter shift-reduce and reduce-reduce conflicts because the grammar does not have ambiguities (i.e., the production rules have been re-written to eliminated ambiguities from the grammar). CUP, an LALR parser generator, on the other hand, will have problems if shift-reduce and reduce-reduce conflicts are encountered.

Bearing in mind the differences between JFlex, CUP and JavaCC, the following section presents additional considerations associated with error recovery.

Error Recovery Strategies

The following section describes the types of error recovery strategies. A review of these strategies is significant since they influence the design considerations for the proposed Java Error Correction Algorithm discussed in this thesis. For example, most compilers implement an error recovery mechanism. However, they do not provide any error correction abilities. The focus of this research is error correction not error recovery.

Panic mode recovery: On discovering an error the input symbols are discarded one at a time until one of a designated set of synchronizing tokens normally delimiters

such as semicolon or the end of the statement is found. This approach is simple, however, it may skip a considerable amount of input.

Phrase level recovery: Perform a local correction by modifying the input stream. These algorithms replace a prefix of the remaining input by some string that allows the parse to continue. For example, replacing a comma by a semicolon, deleting an extra semicolon, or inserting a missed semicolon is the philosophy embedded in these types of recovery algorithms (Burke & Fisher, 1987).

Error productions: Use the language grammar augmented by some error production rules to detect common errors. These extra rules generate erroneous constructs which aid in the recovery process.

Global correction: Still perform a local correction but with global knowledge leading to smallest replacements in order to satisfy the production that failed.

Error recovery in CUP

The error recovery mechanism in CUP is very similar to that of YACC. Both CUP and YACC parser generators support a special error symbol (i.e., denoted as `error`) (Hudson, 1999).

The `error` symbol:

i) plays the role of a special non-terminal which, instead of being defined by productions, matches an erroneous input sequence;

ii) only comes into play if a syntax error is detected; and

iii) is used replace some portion of the input token stream with error if a syntax error is detected and then continue parsing.

For example, we might have productions such as:

```
stmt ::= expr SEMI
| while_stmt SEMI
| if_stmt SEMI
| ...
| error SEMI
;
```

This indicates that if none of the normal productions for **stmt** can be matched by the input, then a syntax error should be declared, and recovery should be made by skipping erroneous tokens. This is equivalent to matching and replacing them with error up to a point at which the parse can be continued with a semicolon.

An error is considered to be recovered if a sufficient number of tokens past the error symbol can be successfully parsed. (The number of tokens required is determined by the *error_sync_size()* method of the parser and defaults to 3).

The parser first looks for the closest state to the top of the parse stack that has an outgoing transition under error. This generally corresponds to working from productions that represent more detailed constructs (e.g., a specific kind of statement) up to productions that represent more general or enclosing constructs (e.g., the general production for all statements or a production representing a whole section of declarations) until a place is reached where an error recovery production has been provided.

Once the parser is placed into a configuration that has an immediate error recovery by popping the stack to the first such state, the parser begins skipping tokens to find a point at which the parse can be continued. After discarding each token, the parser

attempts to parse ahead in the input. It does this without executing any embedded semantic actions. If the parser can successfully parse past the required number of tokens, then the input is backed up to the point of recovery and the parse is resumed normally and will execute all actions. If the parse cannot be continued far enough, then another token is discarded and the parser again tries to parse ahead. If the end of input is reached without making a successful recovery or there was no suitable error recovery state found on the parse stack to begin with, then error recovery fails.

CUP Error Methods

`public void report_error(String message, Object info)`

This method is called whenever an error message is to be issued. The default implementation provides a message which is printed to System.err. Typically, this method is overriden in order to provide a more sophisticated error reporting mechanism.

`public void report_fatal_error(String message, Object info)`

This method is called whenever a non-recoverable error occurs. It responds by calling *report_error()*, then stops parsing by calling the parser method *done_parsing()*, and then throws an exception.

`public void syntax_error(Symbol cur_token)`

This method is called by the parser as soon as a syntax error is detected but before error recovery is attempted. It is invoked automatically by the parser when a production involving the **`error`** symbol is executed.

`public void unrecovered_syntax_error(Symbol cur_token)`

This method is called by the parser if it cannot recover from a syntax error. In the default implementation it calls: *report_fatal_error("Couldn't repair and continue parse", null);*.

`protected int error_sync_size()`

This method is called by the parser to determine how many tokens it must successfully parse in order to consider error recovery successful. The default implementation returns 3 which is the same for YACC and other parser generators.

Error recovery in JavaCC

JavaCC offers the design of the lexical analyzer and parser in a single file. As a result, code that raises errors and exceptions are placed in this file (Norvell, 2004). Whenever the token manager detects a problem, it throws the exception `TokenMgrError`. Whenever the parser detects a problem, it throws the exception `ParseException`. JavaCC supports two default forms of error recovery: Shallow and Deep.

Shallow Error Recovery: Consider the following example:

```
void Stmnt() :
{}
{
  DoWhileStmnt()
|
  WhileStmnt()
}
```

Assume DoWhileStmnts start with the keyword "do" and WhileStmnts start starts with the keyword "while". Suppose the desired error recovery scheme is to recover by skip over all tokens until a semicolon is found when neither DoWhileStmnt nor WhileStmnt can be matched by the next input token (assuming a lookahead of 1). In other words, the next token is neither "do" nor "while". The following code will perform this form of Shallow Error Recovery:

```
void Stmnt() :
{}
{
  DoWhileStmnt()
|
  WhileStmnt()
|
  error_skip_until(SEMICOLON)
}
```

where "error_skip_until" is defined as follows:

```
JAVACODE
void error_skip_until(int kind) {
  ParseException e = generateParseException();
  System.out.println(e.toString());
  Token t;
  do {
    t = getNextToken();
  } while (t.kind != kind);
}
```

"error_skip_until" is no different from other non-terminals in the grammar.

Deep Error Recovery: Consider the same example as used for Shallow recovery:

```
void Stmnt() :
{}
{
  DoWhileStmnt()
|
  WhileStmnt()
}
```

Deep Error Recovery is needed when recovery is required when there is an error deeper into the parse. For example, suppose the next token was "while". As a result, the choice

"`WhileStmnt()`" was taken. However, suppose that during the parse of `WhileStmnt()` an error is encountered. For example, `while (i<10 { i++; }` that is, the ')' is not present. Unfortunately, Shallow recovery will not suffice in these situations -- Deep recovery is required. JavaCC provides deep recovery via the the try-catch-finally construct. A rewrite of the above example for deep error recovery follows:

```
void Stmnt() :
{}
{
  try {
    (
      DoWhileStmnt()
    |
      WhileStmnt()
    )
  catch (ParseException e) {
    error_skip_until(SEMICOLON);
  }
}
```

If there are any unrecovered errors during the parse of DoWhileStmnt or WhileStmnt, then the catch block takes over. Any number of catch blocks may be specified and optionally a finally block (just as with Java errors). What goes into the catch blocks is 100% Java code.

Current State of Intelligent Tutoring Systems

This section presents a review of the current research in Intelligent Tutoring Systems. The review was important as it guided the design and development of the Java Intelligent Tutoring System. The framework for the construction of JITS was based on the widely accepted ACT-R theory of skill acquisition which was developed by a group of computer and cognitive scientists at University of Pittsburg, and Carnegie-Mellon University (Anderson, 1998; Anderson et al., 1995 2). This theory identifies a set of cognitive principles for the development of tutors (Anderson, Boyle, Corbett, & Lewis, 1990; Anderson et al., 1995).

Intelligent Tutoring Systems have undergone significant changes over the years and can be classified into three main categories. The first generation of ITS were basic Computer Aided Instruction (CAI) systems. They presented text or graphics and depending on the student's response, different pages would be shown. Model-tracing ITS were second generation tutors that allow the tutor to follow the student's actions as they work through a problem. The current level of research and development for Intelligent Tutoring Systems is the third generation. These tutors engage in dialog with the student to allow students to construct their own knowledge of the domain. For third generation tutors, interaction with the student is the key element in the design since it is essential to keep the student's attention on-task and as close as possible to the solution path. This has the benefit of minimizing student frustration and reducing off-task activities that do not yield in increased learning (Anderson & Pelletier, 1991).

Heffernan and Koedinger, 2001, state: "We think that if you want to build a good [third generation] ITS for a domain you need to:

i) study what makes that domain difficult, including discovering any hidden skills, as well as determining what types of errors students make;

ii) construct a theory of how students solve these problems (We instantiated that theory in a cognitive model); and

iii) observe experienced human tutors to find out what pedagogical content knowledge they have and then build a tutor model that, with the help of the theory of domain skills, can capture and reproduce some of that knowledge." (p. 24).

Item i) and iii) are well covered by my experience in programming for over 20 years and being a Professor of Computer Science for 10 years at the Sheridan Institute of Technology and Advanced Learning. I was the coordinator of the Computer Science Technology program for several years so I have first-hand knowledge of the curriculum implemented. I have learned and taught over 10 different programming languages at the post-secondary level. I understand Java very well and know the fundamental skills required by students to solve programming problems, and am very aware of the types of errors students make.

Item ii) is supported by ACT-R theory which can be summarized by four principles. ACT-R theory is described in the following section.

ACT-R Cognitive Theory for Developing Tutors

The first principle derived from ACT-R is that it is essential to define the target cognitive model as a set of production rules (Anderson, 1998; Anderson & Pelletier, 1991). Production rules are a set of IF – THEN – ELSE constructs which outline discrete knowledge components which collectively represent the steps required for a student to reach a solution for a problem. A typical ITS may have several hundred production rules to effectively cover the domain and the various states a student may be in within a realm of feasibility and predictability. Heffernan & Koedinger, 2001, reinforce this principle: "Without this [principle] one does not have a well-defined educational goal." (Koedinger, 2001).

The second principle concerns how these production rules are to be communicated to the student (Anderson, 1998). According to ACT-R theory, one cannot directly tell students the underlying rules (Anderson, 1998; Graesser, Person, & Harter, 2001). The goal for ITS is to provide a vehicle by which students construct knowledge for themselves as opposed to having the information told to them (Woolf, Beck, Eliot, & Stern, 2001). ITS need to communicate the production rules to students by providing them with examples that illustrate the rules. As a result, the most effective way for students to construct knowledge is to acquire these rules as a byproduct of problem-solving. JITS is designed to provide various opportunities for students to engage in problem-solving activities for the beginner programmer. This form of experiential learning is an effective way for students to construct knowledge and increase their cognitive abilities (O'Reilly & Munakata, 2000).

The third principle of ACT-R theory is that one wants to maximize the rate at which students have opportunities to form and practice these production rules (Anderson, 1993). Based on other research by ITS researchers, it was shown that what predicts students final achievement is how much practice they have had of these rules and not how that practice occurs (Anderson et al., 1995; Anderson & Pelletier, 1991). Associated with the concept that "practice makes perfect" is the corollary to minimize floundering which is incorporated into many leading-edge Intelligent Tutoring Systems. The basic idea is to reduce student frustration during the problem-solving session and select problems that offer practice on those production rules where students most need practice (Anderson et al., 1995).

The fourth principle of ACT-R cognitive theory for tutoring deals with how to treat errors in student problem solving (Anderson, 1998). Anderson et al. bases this principle on an earlier work in 1990, which states, "people learn best when they generate the answer for themselves rather than are told" (Anderson et al., 1990). However, the consequence of letting people generate their own knowledge is that errors are inevitable. Fortunately, there are four considerations outlined in ACT-R theory that deal with error remediation (Anderson, 1998). First, many errors do not reflect misunderstandings or lack of knowledge; rather the errors are simply unintentional slips. The second consideration is that people learn best when they construct the knowledge themselves. This is analogous to hands-on training as opposed to lecture-based teaching. The third consideration is that a lot of time can be wasted when the student is floundering while trying to solve a problem. This state is called an ***error state*** and is not beneficial for

learning. The fourth consideration is that when students have problems with their knowledge it is more effective to provide another opportunity to learn the correct production. Since the student does not need a deep appreciation of their error, it is not effective for the ITS to expound on it (Heffernan & Koedinger, 2001).

The ACT-R Theory for the development of tutors has led to a standard framework for the design and construction of Intelligent Tutoring Systems. The goal of this framework is to ensure that Intelligent Tutoring Systems will provide rich learning environments for students that will support his/her cognitive development in the specific domain of study in as effective means possible. Many researchers in the area of ITS support the following steps to design and construct an Intelligent Tutoring System.

1) construct the interface;
2) define the production rules;
3) create the declarative instruction;
4) set up the pedagogical agent to knowledge trace, manage the curriculum and engage the student through rich-interaction (Anderson, 1998; Anderson et al., 1990; Heffernan & Koedinger, 2001).

CHAPTER THREE: DESIGN

This chapter presents the design of the Java Error Correction Algorithm in the framework of the design of the Java Intelligent Tutoring System. The design approach for JECA evolved from research in the field of error recovery and the tools available for compiler design. JITS evolved from a collection of work in the field of computer science, cognitive science and AI.

Motivation for the design of the Java Error Correction Algorithm

The initial design of JECA arose from a significant amount of research in the area of knowledge engineering, decision trees and expert systems. For instance, initial research focused on how to identify and correct an error given a simple programming problem using the "`for`" loop construct to calculate the arithmetic sum from 1 to a specified number. Table 2 presents some of the issues with this problem.

Table 2 *Initial design issues for JECA*

Problem:
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
```

Skeleton Program (given to student in Source Code area):

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

    <<Student types code here>>


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

Solution (one of many):

```
public class Summer {
   public static void main(String[] args) {
     int sum = 0;
     int i;
     for (i = 1; i <= 10; i++) {
         sum += i;
     }
     System.out.println("Sum = " + sum);
  }
}
```

Using the arithmetic sum problem described in Table 2, Figure 4 depicts how the expert model's solution may be divided into discrete sections. Based on this, Figure 5 represents the high-level functional decomposition tree for this problem. Figure 6, 7 and 8 present the semantic_decision_tree for this problem. Two methods were proposed to support the feedback mechanism: `general_hint(int context, String snippet)` and `specific_hint(int context, String snippet)`. Although the decision trees isolate the specific area of error within the student's code, additional fine-grained analysis may occur within these methods. These methods are

passed an integer representing the context in which the current programming issue has been identified. The second argument, `snippet`, represents the small portion of code associated with the given `context`.

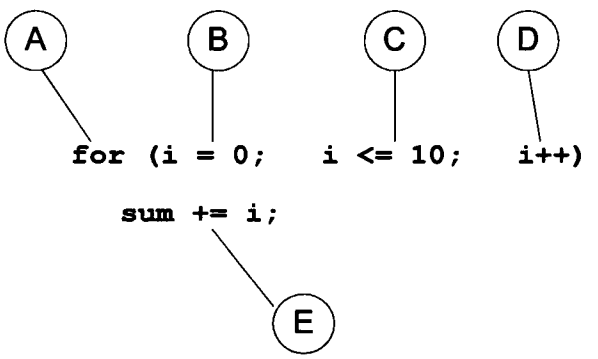


Figure 4. Sub-sections of expert model solution.

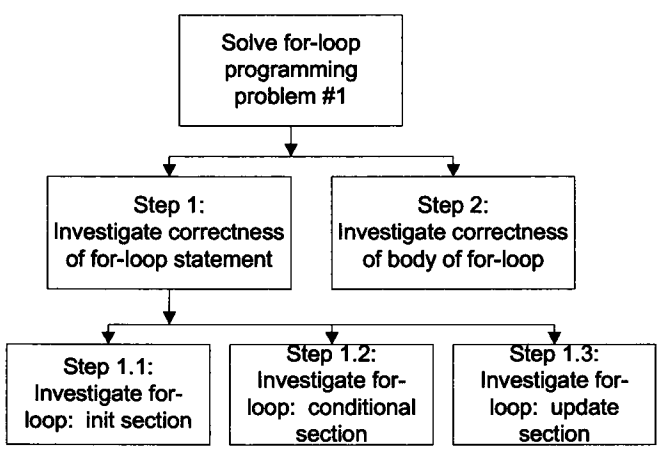


Figure 5. High-level functional decomposition tree.

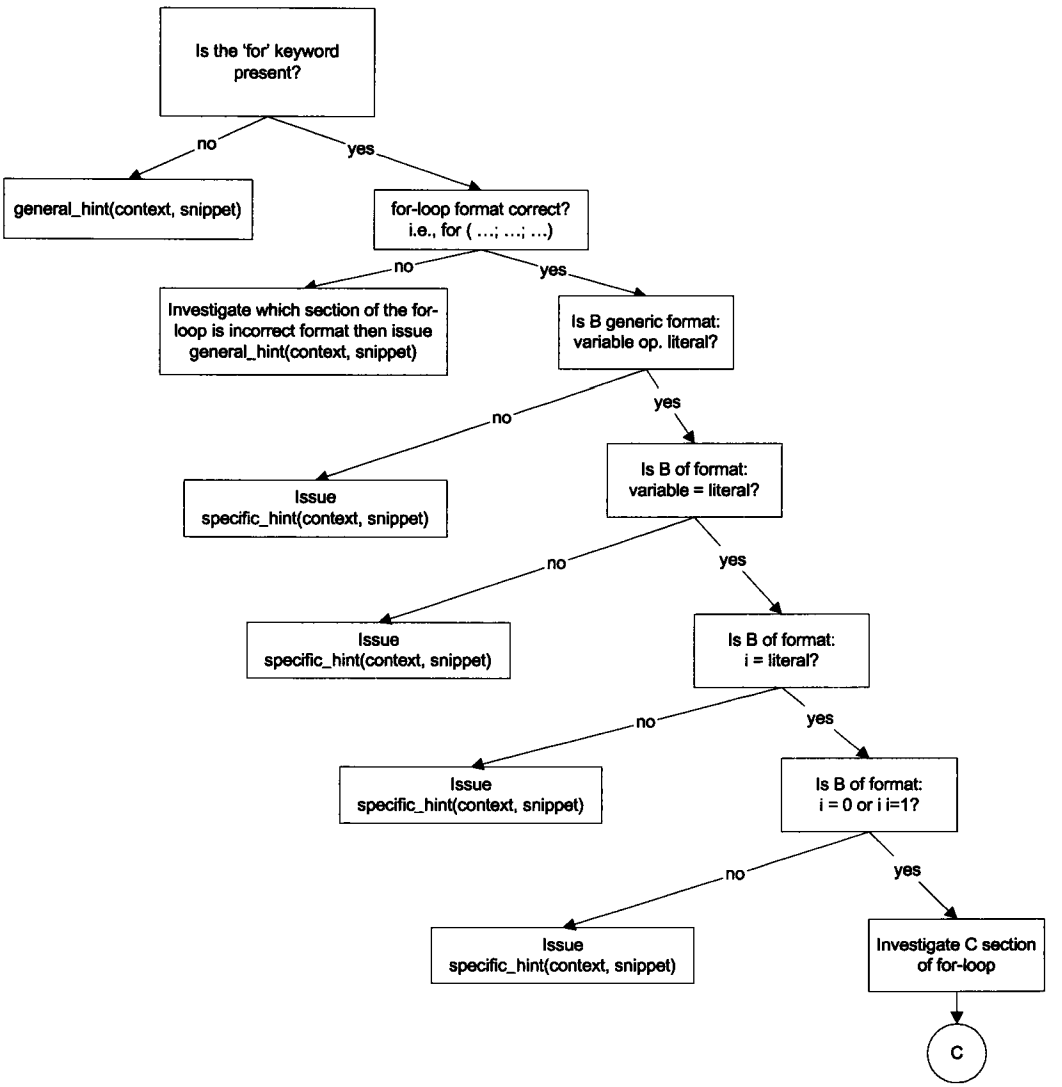


Figure 6. JITS semantic_decision_tree (sections A and B from Figure 4 only).

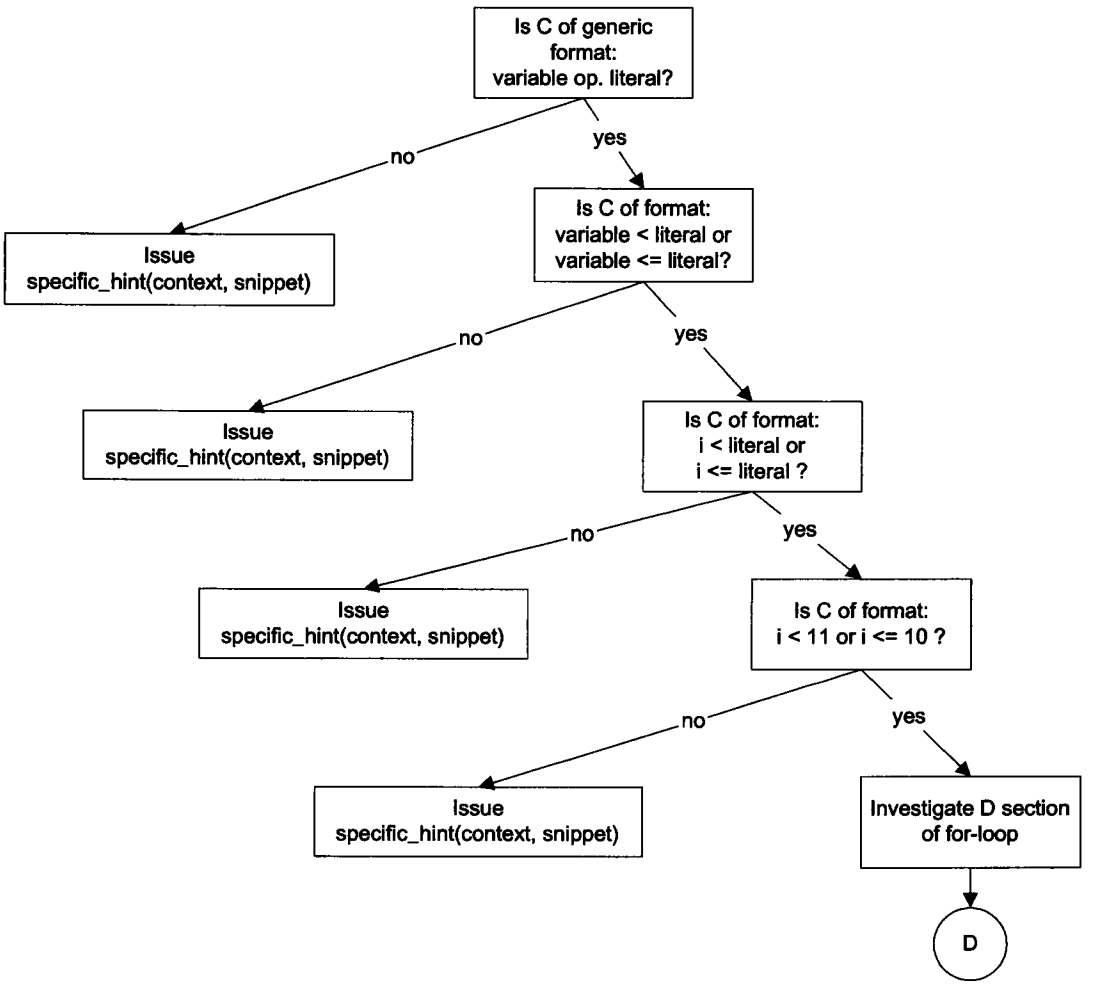


Figure 7. JITS production rules (semantic_decision_tree; section C from Figure 4 only).

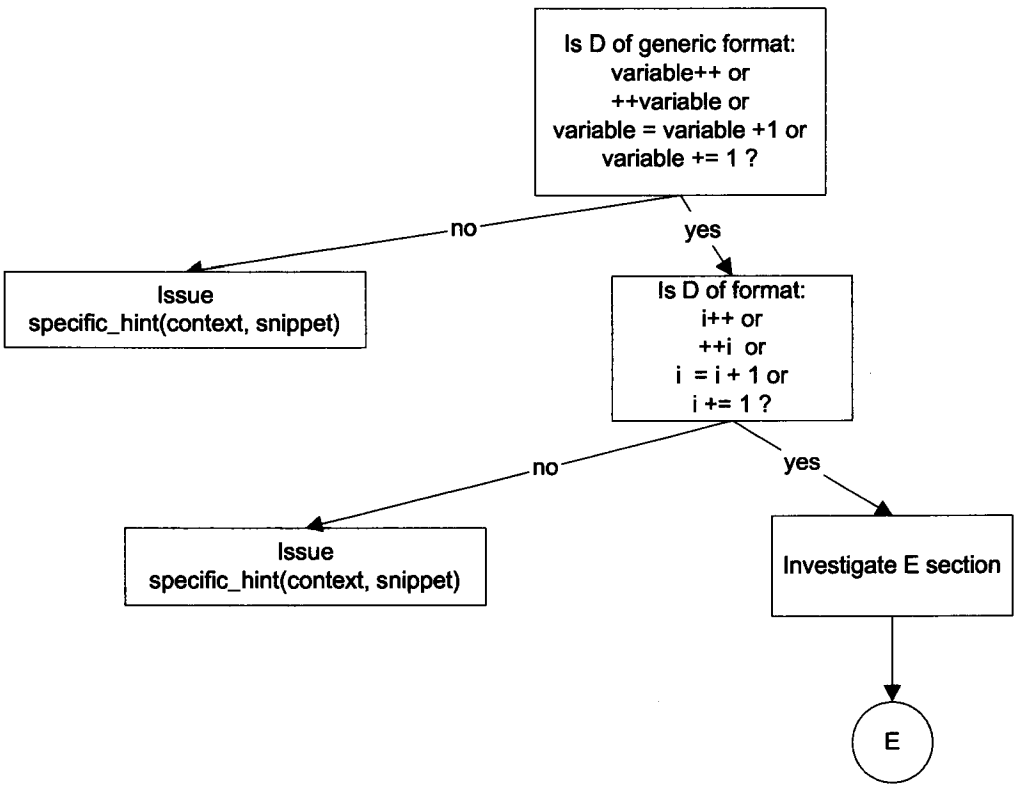


Figure 8. JITS production rules (semantic_decision_tree; sections D from Figure 4 only).

Unfortunately, this representation of the various errors that may occur in such a programming problem requires a huge decision tree. Such a decision tree would still not cover all of the possibilities. To illustrate this issue, Table 3 presents some scenarios of the types of submissions that may be encountered in solving the arithmetic programming problem described in Table 2.

Table 3 *Scenarios representing the various types of code submissions to the arithmetic sum programming problem*

Incorrect response #1 (student response area):
(redeclaration of variable 'i')

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

Incorrect response #2:
(sum does not include last integer (i.e., ' 10')

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

Incorrect response #3:
(sum is 0, as the body of the loop is never executed)

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

Incorrect response #4:
(adding 1 instead of variable 'i': results in sum being lower than expected)

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

Incorrect response #5:
(incorrect formula)

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

Incorrect response #6:
(correct formula, but incorrect incrementing of i)

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

etc.

There are limitless possibilities for student responses and the system cannot simply list incorrect responses coupled with error correction feedback messages. Testing the correctness of a program is not an easy task and cannot be achieved just by giving a set of fixed responses. As a result, attention was turned to various tools as described in the literature review, namely JFlex, CUP and JavaCC as described in the literature review. The researcher had two goals in mind: one, to parse the student's submission more rigorously, and two, to construct an error correction mechanism that would error-correct across all of the Java language. In other words, it would offer meaningful error correction feedback messages not just for the "`for`" loop construct previously described. The design strategy is presented in the following section.

Java Error Correction Algorithm Design

This section describes the design of the Java Error Correction Algorithm. The design arose from research involving decision trees, expert systems, and compiler tools. It became clear after preliminary research that JavaCC provided the best features for the development of an error correction algorithm. JECA is designed to consider three distinct cases:

CASE 1: student enters perfect code and it compiles and runs;

CASE 2: student enters code that needs modification but with JECA changes will compile and run; and

CASE 3: student enters code that needs modification but will not compile regardless of all corrections employed by JECA, however, suggestions are presented to the student to bring the code to a closer state for compilation.

The algorithm used by JECA is presented below.

1. Create a copy of the student's submission (i.e., "modified_source").
2. The scanner examines the student's code and attempts to extract a token. Let *S* be the stream of characters to be validated as a token.
3. A validation process ensues in which comparisons are done using the reserved words and keywords of Java (Table 4), extended keywords (Table 5), and previously declared identifiers.
4. For a given identifier, if the scanner discovers, within a certain threshold, that *S* can undergo transformations to convert *S* into a valid token (i.e., a reserved word or keyword, an extended keyword, or as a previously defined identifier) then it will do so. However, if the scanner determines that *S* is sufficiently different from all of the items previously compared to then it will be left unchanged (i.e., it will remain as a new identifier).
5. Update the modified_source code to reflect these changes and the newly constructed token is submitted to the parser.

6. Repeat 1 through 4 until all input from the student's source code has been processed and the parser has completed the construction of the parse tree representing the modified_source code.
7. Try to parse and compile the modified_source code. If the compilation succeeds then relay the modifications performed to the student in order for them to correct their code and stop processing.
8. If the previous step fails then extract information regarding why it failed and set up a competition of permutated parse trees containing insertions, deletions and replacements at the problem area.
9. Run these permutated trees through the parser. The goal of this stage is to determine if the specific problem where the parse failed has been corrected.
10. Select the "best tree(s)" and compile these. The "best tree" is defined as the tree that allowed the parser to successfully consume the largest number of tokens compared to the other trees in the competition.
11. If one or more of these trees successfully compiles then present this information to the user indicating the changes made to the student's source code.
12. If none of the trees successfully compile then present the information to the student regarding the selection of the best tree.
13. Let the student respond/make corrections to the source code.
14. Repeat the process from 1 to 13.

The algorithm employed by JECA is presented in flowchart form in Figure 9 and Figure 10.

Table 4 *Java Reserved Words and Keywords*

abstract	else	interface	super
boolean	extends	long	switch
break	false ***	native	synchronized
byte	final	new	this
case	finally	null ***	throw
catch	float	package	throws
char	for	private	transient
class	goto *	protected	true ***
const *	if	public	try
continue	implements	return	void
default	import	short	volatile
do	instanceof	static	while
double	int	strictfp **	

Note:

* indicates a keyword that is not currently used

** indicates a keyword that was added for Java 2

*** `true`, `false`, and `null` are reserved words.

Table 5 *Extended Java Reserved Words and Keywords*

Boolean
Character
Number
Byte
Double
Float
Integer
Long
Short
String
StringBuffer

Note:

* this list is a subset of the objects defined in `java.lang.*`

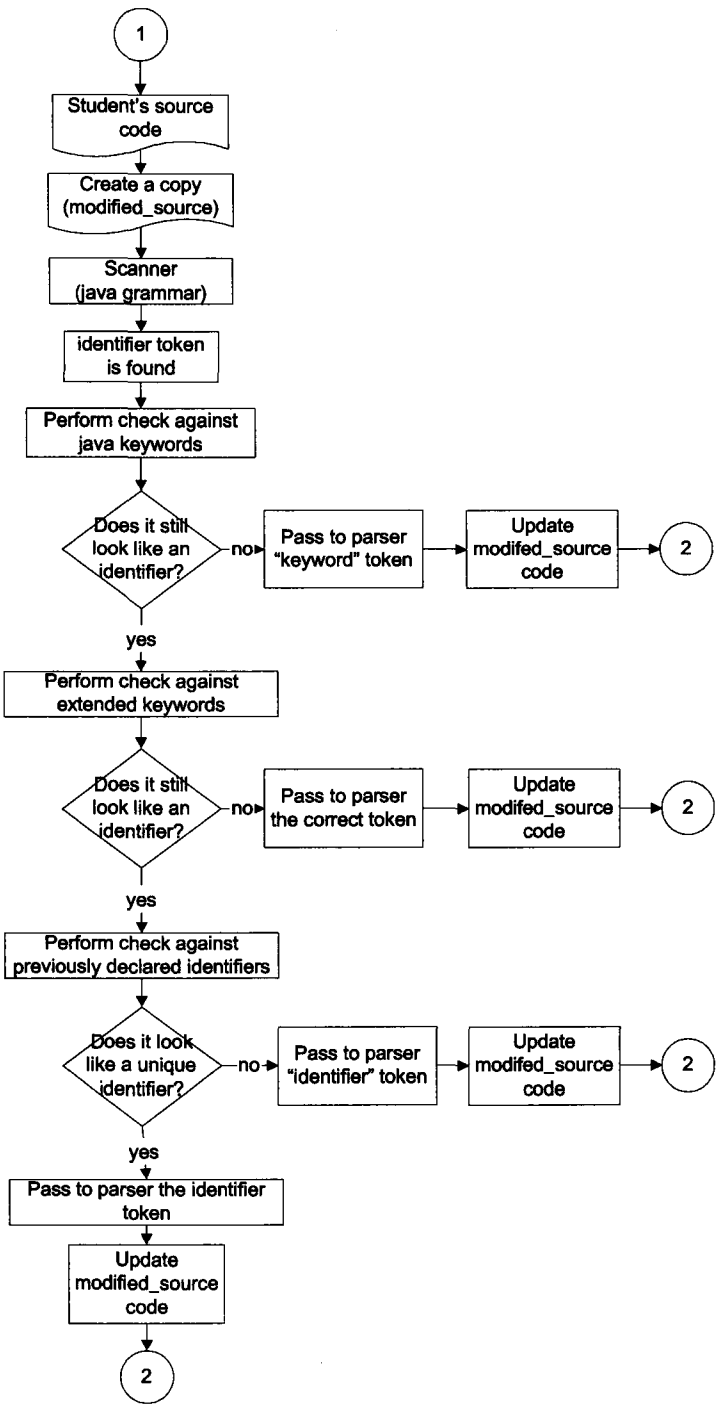


Figure 9. First Component of JECA – scanner correction activities.

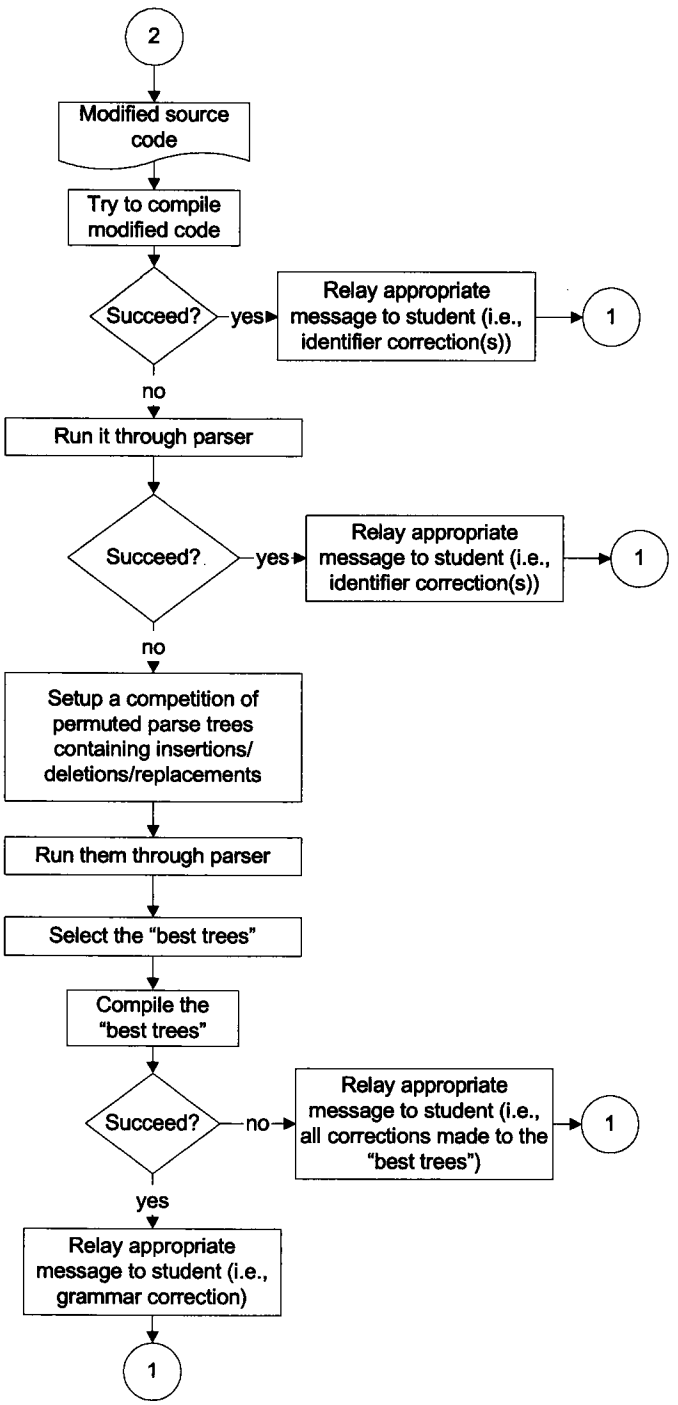


Figure 10. Second Component of JECA – parser correction activities.

Java Intelligent Tutoring System Design

The design of the Java Intelligent Tutoring System heavily relies on JECA to provide the necessary information in order to offer suitable feedback to the student programmer. However, there were a number of factors that were considered in the design of JITS beyond what JECA offered. The two main perspectives that were considered in the design of JITS were both the student's and the instructor's perspective. In order for an ITS to be successful in today's e-learning society, JITS was designed with the following qualities.

Student Perspective

The following qualities were deemed important in the design to satisfy students, were part of the desired list of criteria in the design of JITS:

i. provide an easily understood student-friendly user interface that provides all the necessary features for effective ITS tutoring;

ii. provide access via an ordinary browser;

iii. will not need a high-speed internet connection (i.e., dial-up connection will work fine, thus, students in remote locations have full access to this resource);

iv. process student's code submission and respond quickly to the student;

v. support many students concurrently working with the ITS;

vi. effect interactive, clear and concise with error messages and hints;

vii. track student performance in a database (e.g., ORACLE); and

viii. model the user as s/he works through a problem;

Instructor Perspective

The design of JITS also considered the instructor perspective. The following factors were important in meeting the needs of teachers using this ITS.

i. requires the author of the problem to provide minimal information (e.g., problem statement, program requirements and required output);

ii. the author of the problem does not specify any solutions (this is based on the premise that for a given programming problem there may in fact be numerous solutions);

iii. JITS must be able to recognize a very large number of possible solutions for a particular programming problem;

iv. student performance information should be easily accessible;

v. an instructor-friendly web-based user interface to author problems (i.e., Authoring Tool);

The design of JITS employed the ACT-R theory by following these recommended steps:

1) construct the interface;
2) define the production rules;
3) create the declarative instruction;
4) set up the pedagogical agent to knowledge trace, manage the curriculum and engage the student through rich-interaction (Anderson, 1998; Anderson et al., 1990; Heffernan & Koedinger, 2001).

From a pedagogical perspective, the framework depicted in Figure 11 was used.

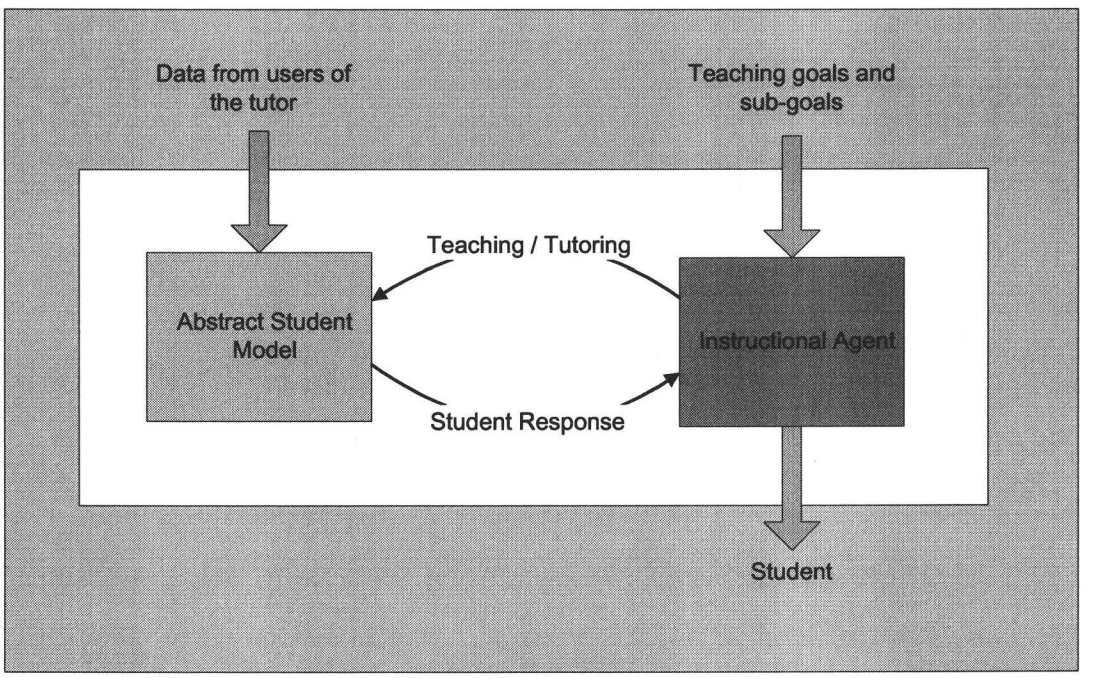


Figure 11. JITS Tutoring Framework.

CHAPTER FOUR: IMPLEMENTATION

Java Error Correction Algorithm (JECA) Implementation

The core module of the Java Intelligent Tutoring System is the Java Error Correction Algorithm (JECA). The first component of JECA involves scrutinizing the identifiers that the scanner has tokenized by comparing them to keywords, reserved words, extended 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 (Burke & Fisher, 1987). This algorithm is explained in greater depth in the following sections.

First Component of JECA: Error Recovery in the Scanner (Lexical Analyzer)

It is sometimes desirable to change what the scanner has interpreted to a single Java keyword. The reserved words and keywords in the Java programming language is presented in Table 4. As an 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 proposed error recovery algorithm, 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 such as "I found an 'Int'. Would you like to replace it with 'int'? (y/n)" In this fashion, the student of the system is fully aware of all changes that are taking place on the submitted code. In other words, all changes are made explicitly known to the user. This philosophy is different from other compiler designs that make changes to the source program without notifying the user (Fischer & LeBlanc, 1991; Sykes & Franek, 2003).

For example, an analogy is found in the C programming language. Given a simple program like:

```
main() {
     return 0;
}
```

may be interpreted by a compiler as:

```
int main() {
     return 0;
}
```

The compiler implicitly puts in the default type '`int`' during compilation. Such implicit changes can be misleading to the user (Fischer & LeBlanc, 1991). JECA, on the other hand, does not do any implicit changes to the code. All code changes are overt. A supporting mechanism used to do this is depicted in Figure 12.

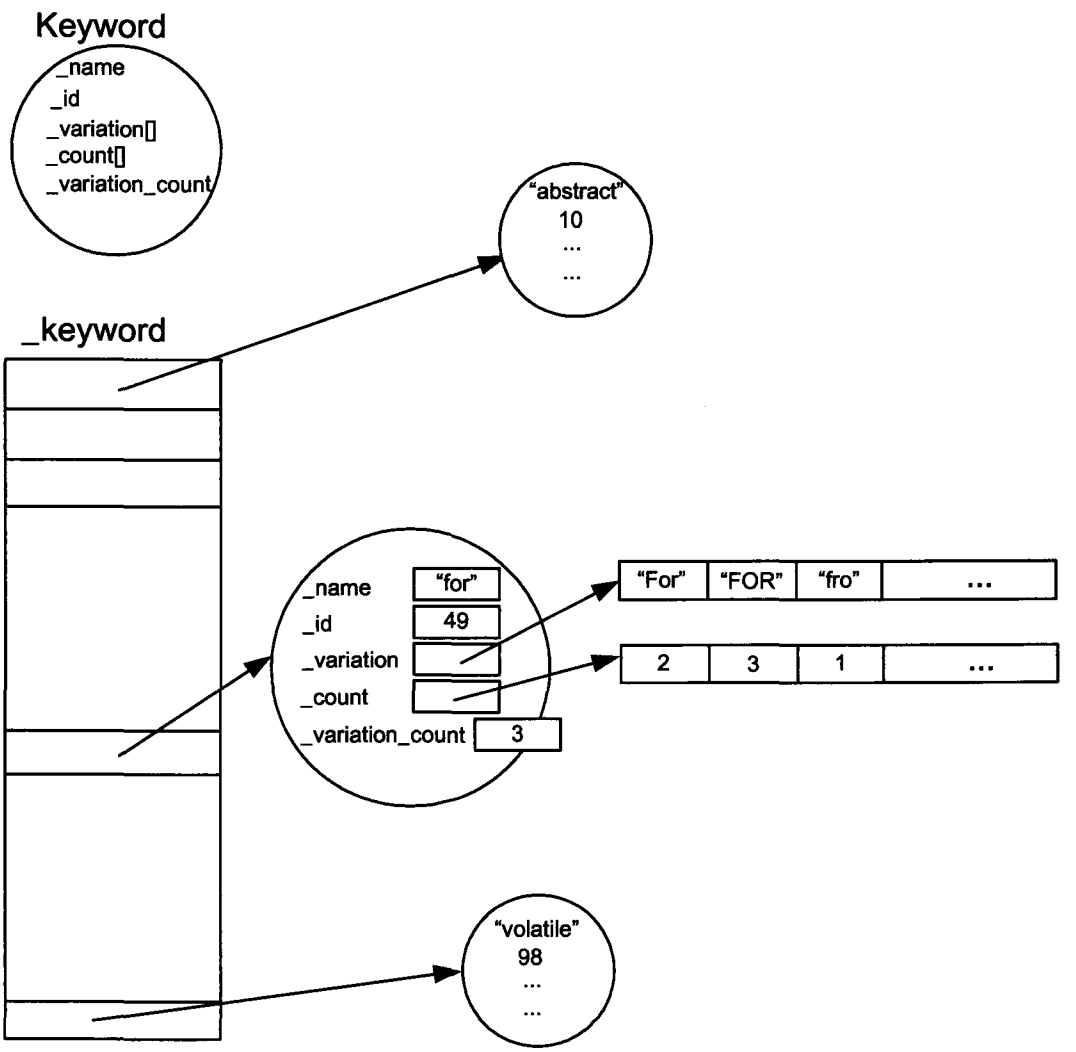


Figure 12. Keyword object and _keyword data structure.

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 _variation_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. Similar data structures are implemented for Extended_Keywords, and Identifiers in order to record information regarding these types of data. This information is gathered during the lexical analysis phase by JECA.

Given a lexeme that has currently been classified as an identifier token, the objective is to analyze this lexeme and determine if it should remain as an identifier or be classified as a different type of token. The algorithm includes a reference to the Edit_Distance object that has a method to determine the edit distance between two strings. For example, given the strings, "while" and "wiles", the edit distance is 2 (i.e., a count of 1 for the missing character 'h', and 1 for the additional character 's'). The algorithm for this identifier-classification 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 the refinement collection and determine if it
should be considered a keyword, extended_keyword, or as a new
identifier
```

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 lexical analyzer (i.e., TokenManager in JavaCC) returns the appropriate Token to the parser. A figure of the BestMatch object is presented in Figure 13.

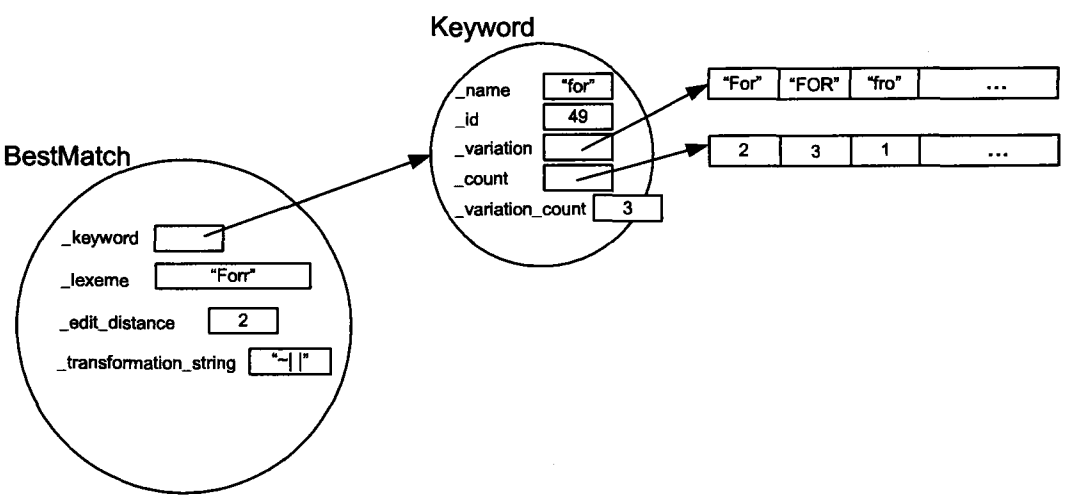


Figure 13. BestMatch object – used for the refinement process in determining an identifier or a keyword.

A member of the BestMatch object is _transformation_string. This member receives this 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 (Sykes & Franek, 2003).

The cost for an insertion, deletion, transposition, or character change is 1. Figure 14 depicts a transformation string given two strings "`Forr`" and "`for`". The algorithm is quite flexible and can be easily modified to accommodate various scenarios. For example, the edit distance in Figure 14 could be 2 (i.e., case-mismatch 'F' and an additional 'r'). It could also configured to produce an edit distance of 1.5 (i.e., case-mismatch = 0.5 and 1 for the additional 'r') or any other cost depending on setting some switches. The rationale behind this is based on the premise that the algorithm should draw close relationships between strings that have the correct sequence of characters but may not have the correct case. Researchers in the area of education and psychology believe this concept is pedagogically sound (O'Reilly & Munakata, 2000). A student who uses "For" instead of "for" has a clearer conceptual understanding of the "for loop" construct than a student who uses "Fore" for instance. These different cognitive models are reflected in the algorithm.

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

Figure 14. BestMatch member contains the Transformation string from Edit_Distance algorithm.

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 (Burke & Fisher, 1987; Fischer & LeBlanc, 1991). 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 (Burke & Fisher, 1987). 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 (Burke & Fisher, 1987).

The Burke-Fisher Error Recovery algorithm uses 2 stacks, *current* and *old*, and a *queue* of *k* tokens (Burke & Fisher, 1987). *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*. Figure 15 and Figure 16 depict an example using the Burke-Fisher error recovery algorithm.

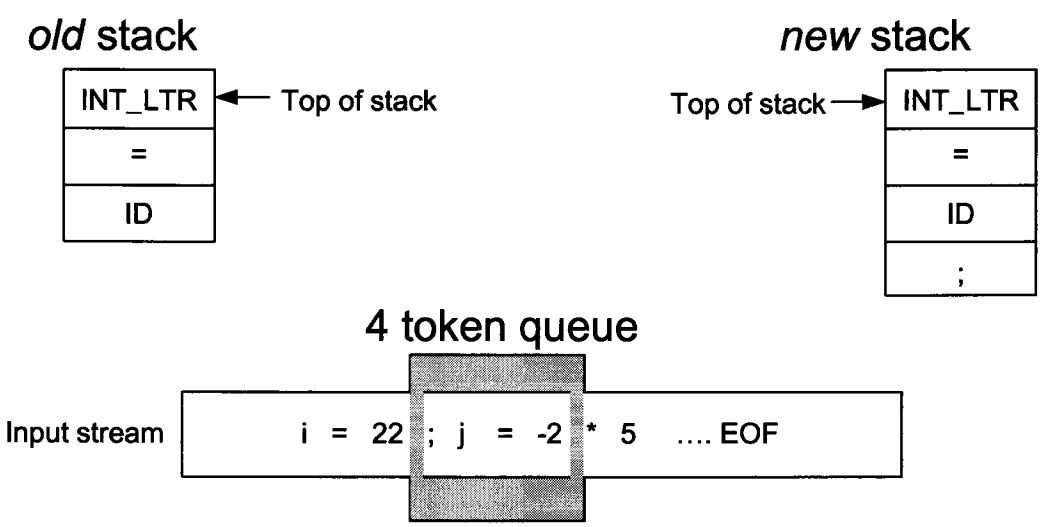


Figure 15. Burke-Fisher error correction algorithm with a 4-token queue in the middle of processing a statement production.

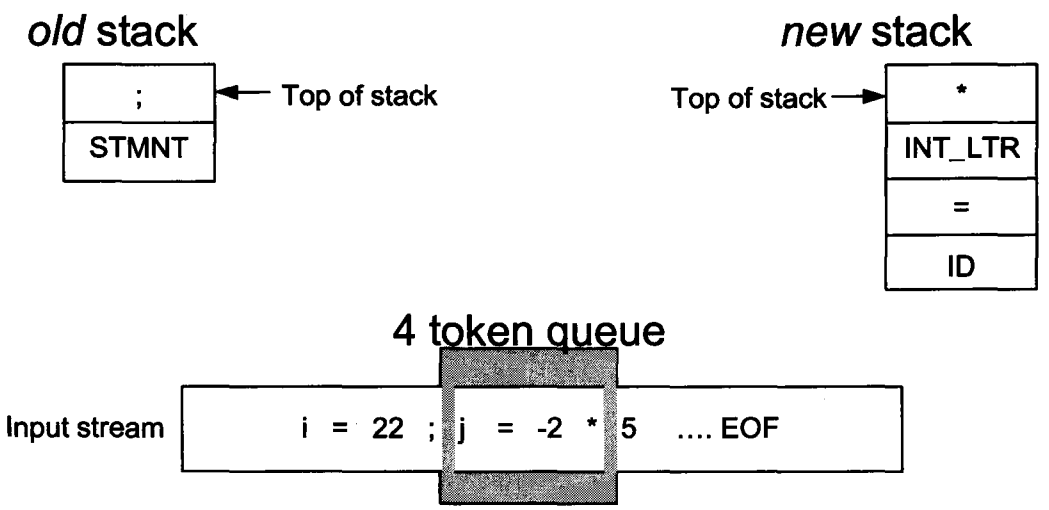


Figure 16. Burke-Fisher error correction algorithm with a 4-token queue completing the processing of a statement production and commencing a new production.

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 intermediate 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 byte 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. 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 error correction 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 such that the parse tree(s) 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 6 depicts this internal JECA functionality. Please note the student does not see any of these computations.

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 (Anderson et al., 1995; Sykes, 2003). It is important that the student programmer does not become overwhelmed by the number of error messages produced by compilers when errors occur (Graesser et al., 2001; Koedinger, 2001). 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 (O'Reilly & Munakata, 2000).

Table 6 *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;
   }
}
```

Java Intelligent Tutoring System Implementation

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 (Pawlan, 2004). It is scalable, platform-independent, and lightweight (Pawlan, 2004). 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. Figure 17 presents a pictorial view of the JITS multi-threaded distributed Web-based Infrastructure.

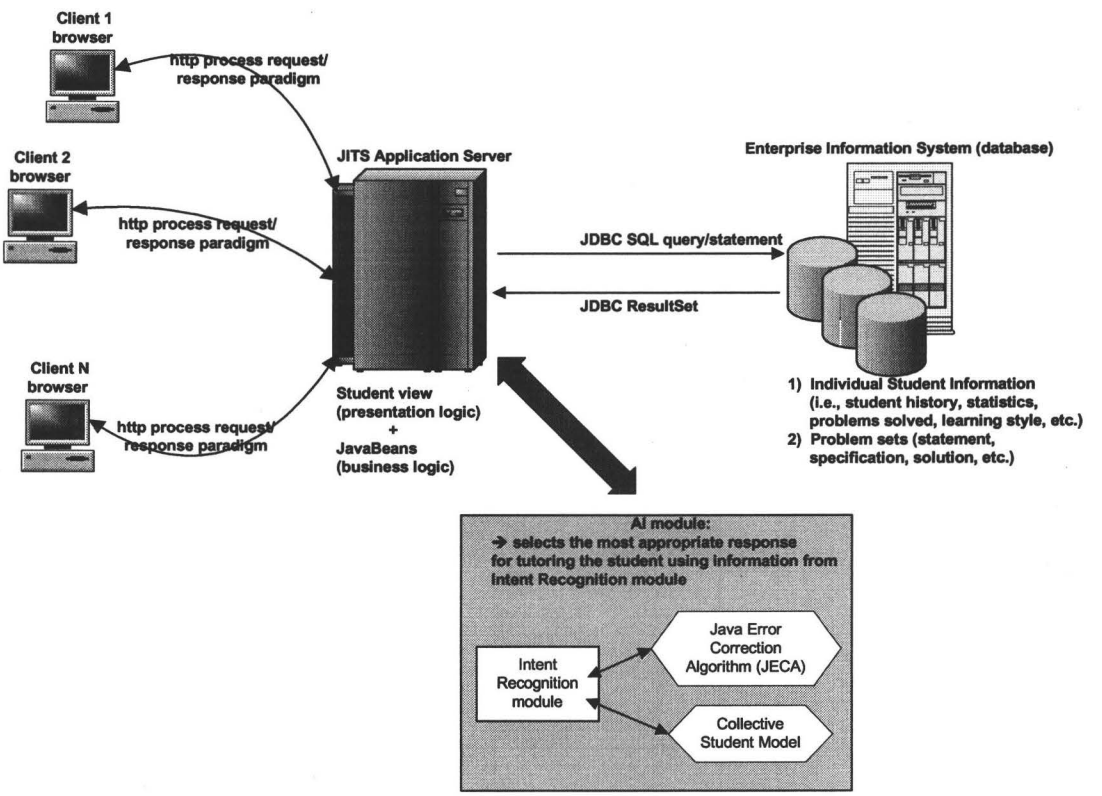


Figure 17. JITS multi-threaded distributed web-based infrastructure.

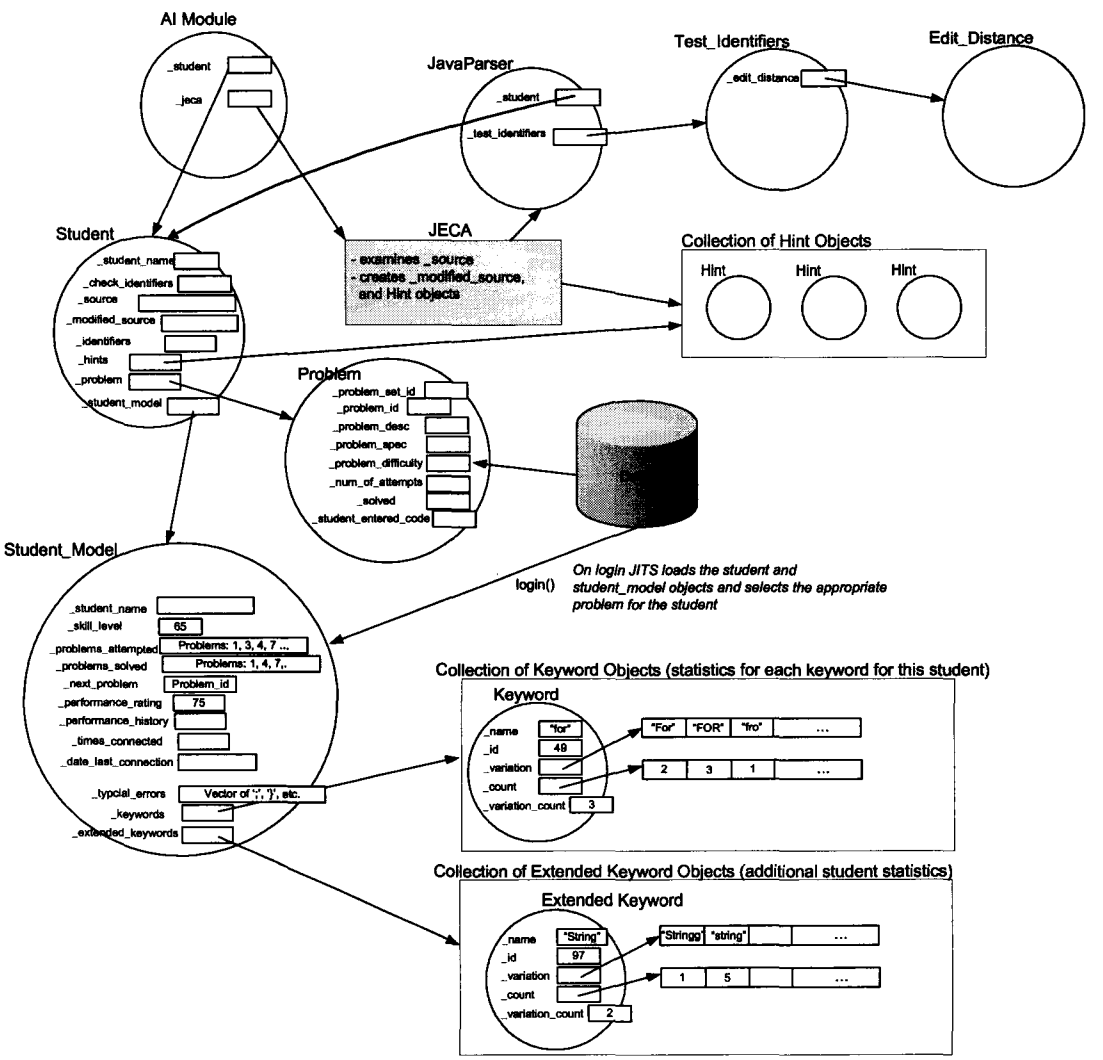


Figure 18. JITS student model and related modules.

Human-Computer Interaction

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.). The JITS login screen and user interface is shown in Figure 19 and Figure 20 respectively.

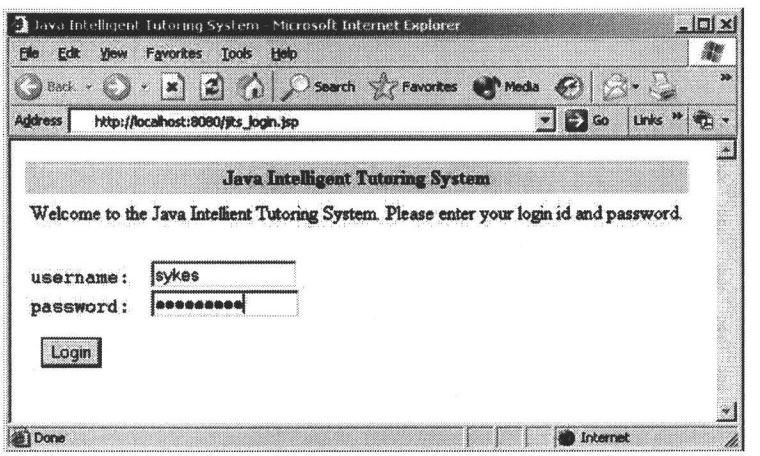


Figure 19. JITS login screen.

Students are presented with a problem, the problem specification, the skeleton code, the code editor, and a number of buttons with which to interact with the tutor. The student types in his/her solution in the Source Code Area (see Figure 20) 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 "View Top Hint" or "View All Hints". The hints are dynamically generated based on the problem details and the student's submission.

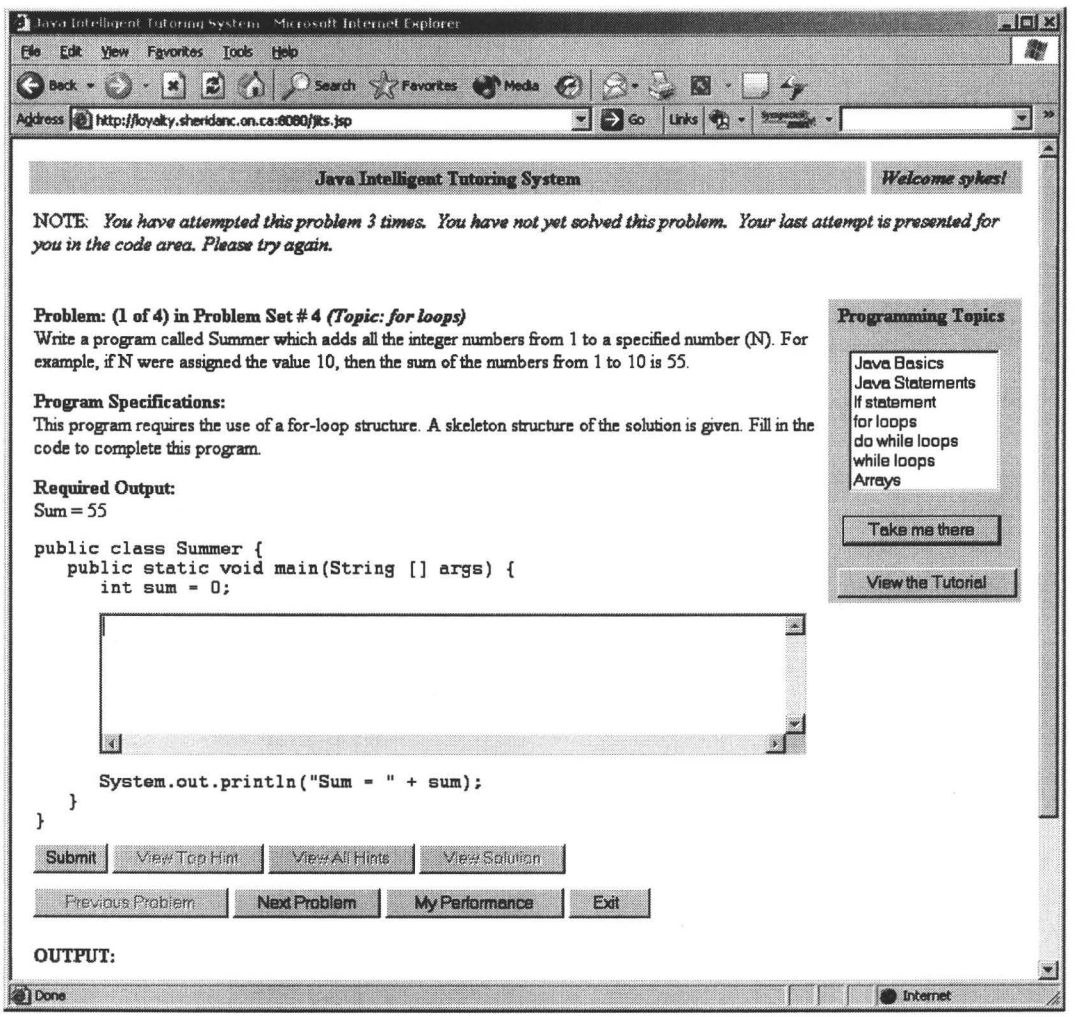


Figure 20. JITS User Interface.

In the Java Intelligent Tutoring System teachers are not required to submit solutions during problem authoring. This is based on the premise that given virtually all programming problems there are potentially limitless solutions. Supplying only one solution for a given programming problem is not an acceptable approach. As a result, a Collective_Student_Model representing the sum knowledge of all students was designed

and developed. This Collective_Student_Model analyses all student's submissions and extracts those which are solutions to the particular problem the student is currently working on. The AI_Module uses the information in Collective_Student_Model to determine appropriate feedback. Table 7 shows a small example illustrating this additional functionality. At any time, the student can see their performance by pressing the "My Performance" button. The results are displaying indicating the Problem Set, Problem #, Solved, Solution Viewed, and for comparison, the Average Student Performance. Table 8 depicts the output from the "My Performance" button.

Table 7 *"View Solution" presenting solutions for the current problem*

Solution 1:

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

Solution 2:

```
for (int i = 1; i<11;i++)
{
   sum = sum + i;
}
```

Solution 3:

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

Table 8 *JITS results from student pressing the "My Performance" button.*

My Performance:

Problem Set	Problem #	Solved	Solution Viewed	Average Student
1	1	**No.** 4 attempts so far.	***Yes.***	2 attempts to solve.
1	2	**No.** 2 attempts so far.	***Yes.***	2 attempts to solve.
1	3	**No.** 4 attempts so far.	***Yes.***	2 attempts to solve.
3	1	**No.** 1 attempt so far.	**No.**	1 attempt to solve.
3	2	**No.** 1 attempt so far.	**No.**	1 attempt to solve.
4	1	***Yes.*** It took 5 attempts.	**No.**	2 attempts to solve.
4	2	**No.** 3 attempts so far.	**No.**	3 attempts to solve.
4	3	**No.** 2 attempts so far.	**No.**	6 attempts to solve.

JITS contains the following programming topics:

i) Java Basics;
ii) Java Statements;
iii) if statement;
iv) for loops;
v) do while loops;
vi) while loops;
vii) Arrays

Each programming topic corresponds to "Problem Sets" which contain many "Problems". All of the information on the web page is dynamically constructed using

Java ServerPages™ technology. Information is extracted via JDBC, from an ORACLE database schema and embedded into the JITS web page. The schema is presented in Table 9.

Table 9 *JITS ORACLE schema tables*

```
CREATE TABLE PROBLEM_SETS (
      problem_set_id      NUMBER(3),
      problem_set_title   VARCHAR2(30),
      problem_set_desc    VARCHAR2(400),
);

CREATE TABLE PROBLEMS (
      problem_set_id            NUMBER(3),
      problem_id                NUMBER(3),
      problem_desc              VARCHAR2(400) NOT NULL,
      problem_spec              VARCHAR2(400) NOT NULL,
      problem_output            VARCHAR2(50),
      template_top_section      VARCHAR2(400),
      template_bottom_section   VARCHAR2(400),
      problem_difficulty        VARCHAR2(20),
      problem_keywords          VARCHAR2(200),
      picture                   LONG RAW,
);

CREATE TABLE STUDENTS (
      student_name              VARCHAR2(30),
      student_password          VARCHAR2(15),
      problem_set_id            NUMBER(3),
      problem_id                NUMBER(3),
      skill_level               NUMBER(3),
      performance_rating        NUMBER(3),
      performance_history       VARCHAR2(2000),
      times_connected           NUMBER(5),
      date_last_connection      VARCHAR2(30),
      picture                   LONG RAW,
);

CREATE TABLE STUDENT_PROBLEMS (
      student_name              VARCHAR2(30),
      problem_set_id            NUMBER(3),
      problem_id                NUMBER(3),
      number_of_attempts        NUMBER(3),
      solved                    CHAR(1),
      students_solution         VARCHAR2(500),
      solution_date             VARCHAR2(30),
);
```

Hint Generation

An additional design consideration is the categories of hints that are generated by JECA for JITS. There are a number of different categories of hints that may be created as a result of the student's code submission. They are presented in Figure 21.

```
KEYWORD_REPLACEMENT_HINT = 1;
EXTENDED_TYPE_REPLACEMENT_HINT = 2;
IDENTIFIER_REPLACEMENT_HINT = 3;
GRAMMATICAL_HINT = 4;
CLOSE_BUT_LOGIC_ERROR = 5;
SUCCESSFULLY_SOLVED_PROBLEM = 6;
GENERAL_HINT = 7;
OTHER_TYPE_OF__HINT = 8;
```

Figure 21. 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. The last two types of hints 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. If the student's code compiles but produces output that is not the same as the required output, as specified in the problem statement, the CLOSE_BUT_LOGIC_ERROR is used. When the student solves the problem the SUCCESSFULLY_SOLVED_PROBLEM hint is used. Lastly, 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 22. 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`.

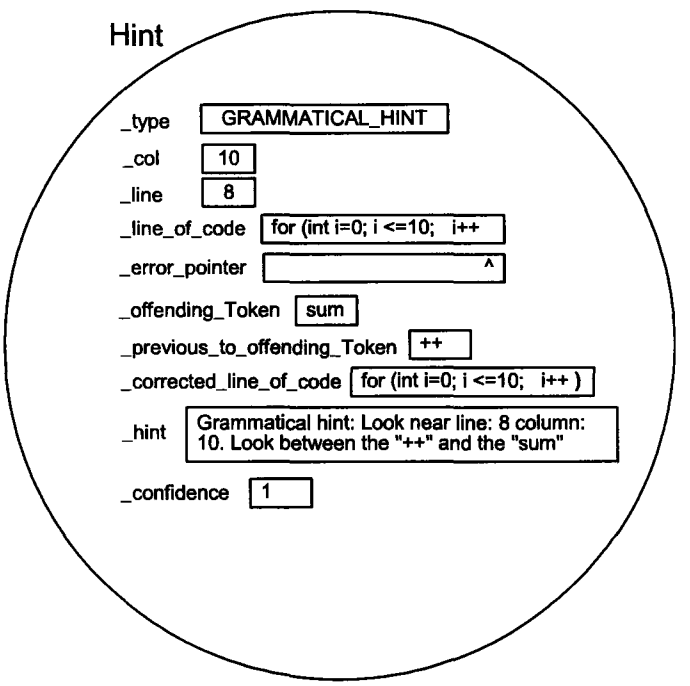


Figure 22. A JECA Hint object representing a grammatical error.

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

Given the source program depicted below:

```
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 23. 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 24. 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 23 , focusing only on the area where the student enters code in the "source code area" (see Figure 20).

Table 10 presents the dialogue between JITS and the student.

Table 10 *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 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 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++    {
                      ^
```

Missing ")"

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 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, or 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
```

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

User Modeling

JITS tracks a great deal of information about the student as s/he works on programming problems in the system. The ultimate goal of gathering this information is to more closely model the student and to more effectively assist the student during the tutoring process. The following list describes the information tracked by JITS:

1. time and date when a student logs onto JITS;
2. the number of times the student has connected to JITS;
3. every code submission the student makes on a problem;

4. the number of attempts for each problem the student has tried;
5. the student's solution to a problem;
6. the number and type of misconceptions involving keywords, extended keywords, and identifiers are recorded (e.g., "For", "fro" instead of "for", etc.);
7. whether or not the student pressed the "View Solution" button for a problem;
8. student movement through each Problem Sets;
9. student movement to a different topic (i.e., the types and difficulty of problems the student attempts is recorded);

Collectively, this information allows JITS to model the student and effectively engage the student in the tutoring process. When a student exits the system, the next time the student starts JITS, the system brings the student back to the exact state when s/he left. That is, the problem and code the student was working on is presented to the student. Table 11 and Table 12 depict some of the student tracking information.

Table 1 *Sample database student tracking information indicating number of attempts, solved (true/false), and student's solutions*

STUDENT_NAME	PROBLEM_SET_ID	PROBLEM_ID	NUMBER_OF_ATTEMPTS	SOLVED	STUDENTS_SOLUTION
dav_sem3_student10	4	1	6	F	
dav_sem3_student16	4	1	8	T	for (int i = 1;i <= 10;i++) sum = sum + i;
dav_sem3_student16	4	2	0	F	
dav_sem3_student20	4	1	2	F	fdsf
dav_sem3_student3	4	1	3	T	for (int i = 0; i <= 10; i++) sum = sum + i;
dav_sem3_student3	4	2	1	T	for (int i = 4; i > 1; i--) fact = fact * i;
dav_sem3_student3	4	3	16	F	for (int i = 1; i = 500; i = i + 2) total = total + i;
dav_sem3_student5	4	1	0	F	
dav_sem3_student6	4	1	1	T	for (int i=1; i<=10; i++) sum += i;
dav_sem3_student6	4	2	1	T	for (int i=1; i<=4; i++) fact *=i;
dav_sem3_student6	4	3	10	F	for (int i=0; i<=500; i=i+1) total +=i-1;
dav_sem3_student7	4	1	1	T	for (int i=1; i<=10; i++) sum +=i;
dav_sem3_student7	4	2	1	T	for(int i=1; i<=4; i++) fact *=i;
e	4	1	4	T	for (int i=0; i<=10; i++) sum = sum +i;

Table 2 *Sample database student tracking information indicating current problem set, problem id, performance rating, skill level, number of times connected to JITS, and the date of last connection.*

STUDENT_NAME	PROBLEM_SET_ID	PROBLEM_ID	SKILL_LEVEL	PERFORMANCE_RATING	PERFORMANCE	TIMES_CONNECTED	DATE_LAST_CONNECTION
e	4	1	1	81	null	12	Fri Jun 04 15:56:56 EDT 2004
dav_sem3_student1	4	1	1	1		0	
dav_sem3_student10	4	1	1	1	null	2	Wed Jun 02 12:54:26 EDT 2004
dav_sem3_student11	4	1	1	1		0	
dav_sem3_student12	4	1	1	1			0
dav_sem3_student13	4	1	1	1			0
dav_sem3_student14	4	1	1	1			0
dav_sem3_student15	4	1	1	1			0
dav_sem3_student16	4	1	1	1	null	1	Wed Jun 02 13:03:10 EDT 2004
dav_sem3_student17	4	1	1	1			0
dav_sem3_student18	4	1	1	1			0
dav_sem3_student19	4	1	1	1			0
dav_sem3_student2	4	1	1	1			0
dav_sem3_student20	4	1	1	1	null	2	Fri Jun 04 15:54:40 EDT 2004
dav_sem3_student21	4	1	1	1			0
dav_sem3_student22	4	1	1	1			0
dav_sem3_student23	4	1	1	1			0
dav_sem3_student24	4	1	1	1			0
dav_sem3_student25	4	1	1	1			0
dav_sem3_student26	4	1	1	1			0
dav_sem3_student27	4	1	1	1			0
dav_sem3_student28	4	1	1	1			0
dav_sem3_student29	4	1	1	1			0
dav_sem3_student3	4	1	1	83	null	1	Wed Jun 02 12:56:44 EDT 2004
dav_sem3_student30	4	1	1	1			0
dav_sem3_student31	4	1	1	1			0
dav_sem3_student32	4	1	1	1			0
dav_sem3_student33	4	1	1	1			0

There are other structures in JITS that support the student as s/he work through the problems. A Help window is available that displays the organization of JITS from a user interface perspective. It shows the various sections of JITS including the "Code Area", the various interactive buttons, and the "Output" area. Figure 25 depicts the Help window.

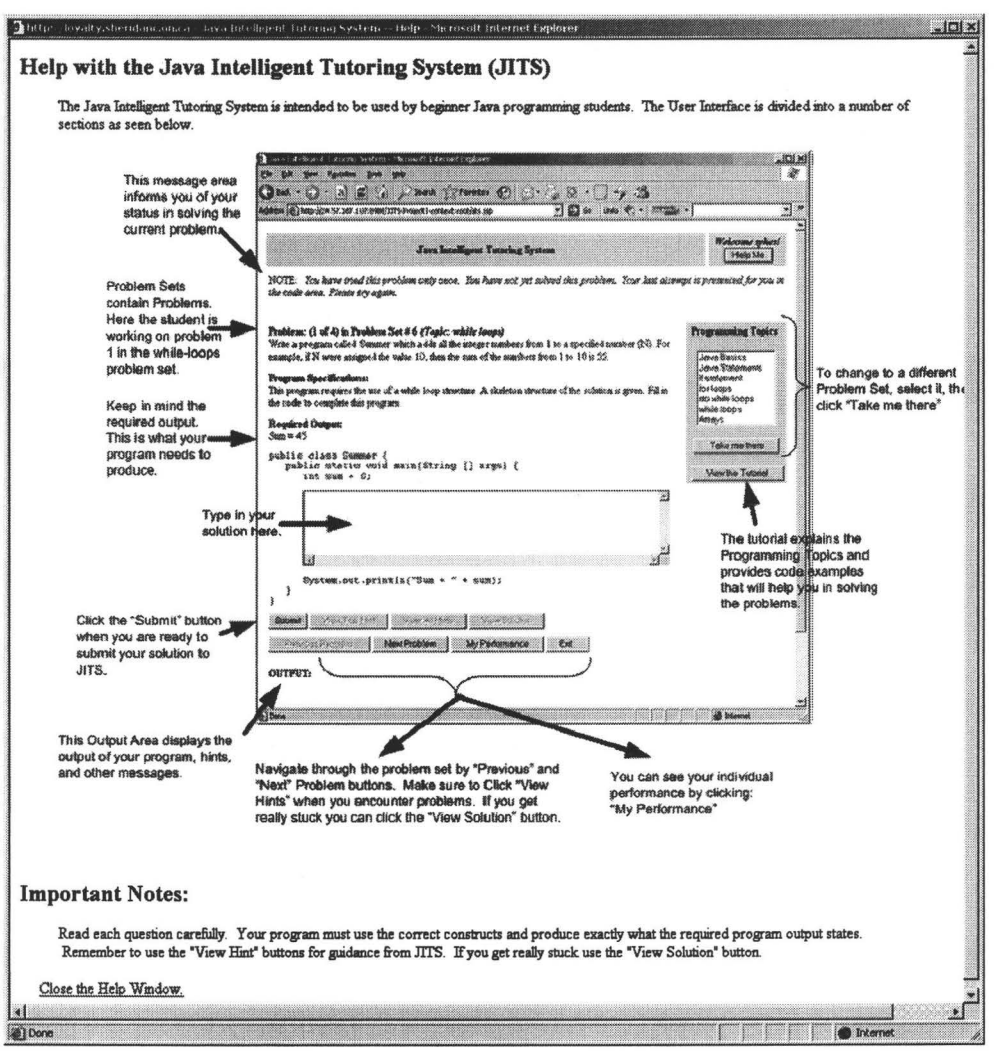


Figure 25. Java Intelligent Tutoring System Help window.

A Tutorial was developed that presents important aspects for all of the programming topics. The description of programming constructs, syntax, and examples are included in this concise tutorial window. Figure 26 depicts this information.

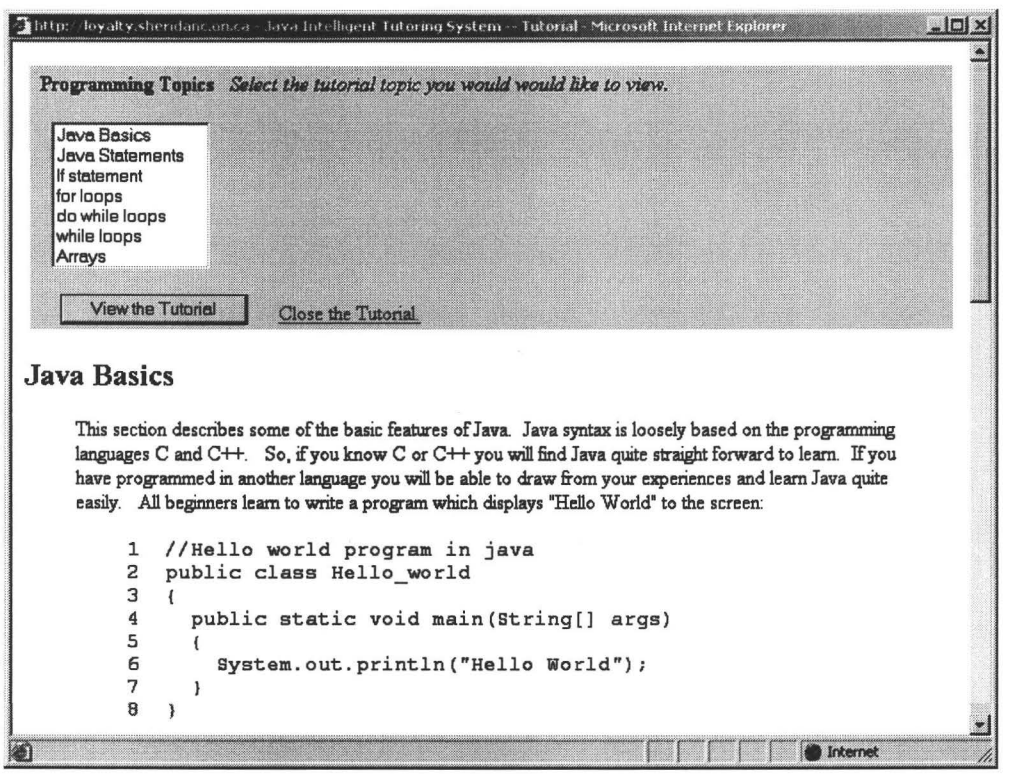


Figure 26. Java Intelligent Tutoring System Tutorial window.

Additionally, consideration for visually-oriented students resulted in the development of the ImageViewer window. Every problem has the option of containing a picture to elucidate the problem description. Like all the information displayed in the user interface, JITS extracts the details from the database and dynamically inserts it in the web pages via Java ServerPage™ technology. Figure 27 depicts the JITS ImageViewer.

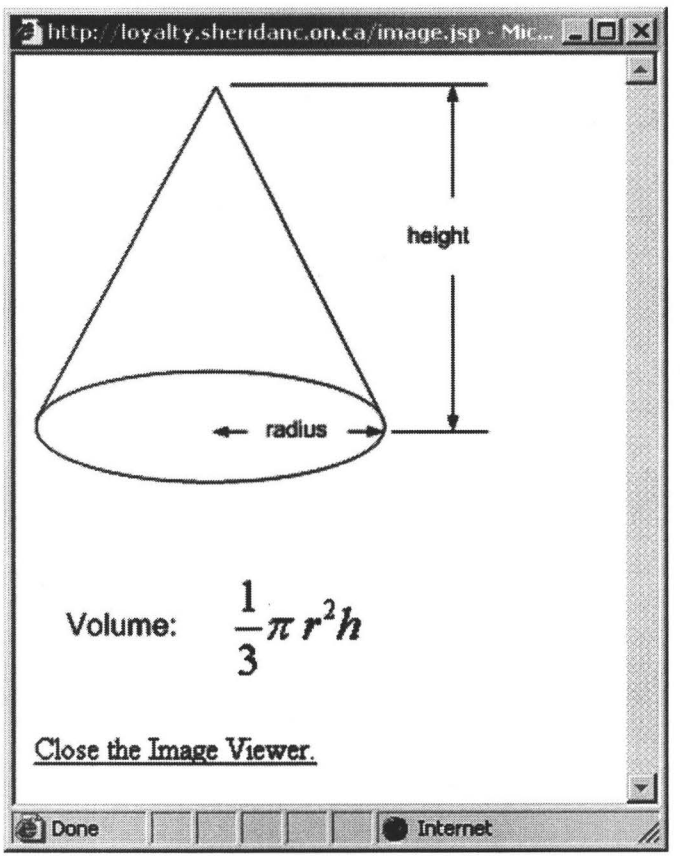


Figure 27. Java Intelligent Tutoring System Image Viewer window.

CHAPTER FIVE: SUMMARY, CONCLUSIONS AND RECOMMENDATIONS

Summary

This thesis focused on the design and development of an error correction algorithm in the specific context for use in the Java Intelligent Tutoring System. The Java Error Correction Algorithm was designed for small Java programs created by first year College and University students with little or no programming experience. JECA attempts to determine the "intent" of the student's program by carefully analyzing the student's code. JECA makes "intelligent" changes to the student's code during lexical analysis and parsing. During lexical analysis, identifiers are carefully scrutinized and reclassified as keywords, extended keywords, or previously declared identifiers, if appropriate. During the second phase, JECA performs error corrections if the student's modified code does not compile. During this phase, a collection of parse trees are constructed containing permutations including insertions, deletions, and replacements. A competition is arranged and the best tree(s) are selected for further analysis with the objective to bring the students' code closer to a state of successful compilation. Behind the scenes, JECA compiles and runs these trees to gather more information about the changes that have been made to the student's program. Unlike other error recovery strategies, JECA does not hide any changes that have been made to the student's code. On the contrary, JECA makes all the changes known to the student. As a result, JECA produces a significant amount of information that is relayed to the Java Intelligent Tutoring System that are used in the tutoring process. This has many benefits in terms of knowledge and skill development for students the JITS system.

This thesis focused on JECA, however, to ensure contextual relevance and significance, the Java Intelligent Tutoring System was included. JITS is implemented using advanced e-learning technologies and its multi-threaded distributed architecture makes JITS scalable, robust and easy to maintain. Through the use of Java ServerPages™, and JavaBeans™, all processing is done at the middle-tier level. The Model-View-Controller (MVC) design pattern was used to implement JITS. All content is dynamically extracted from an ORACLE database via JDBC and placed into the appropriate Java ServerPage™. From a pedagogical perspective, JITS supports personalized student development by modeling every student in the system. JITS also enhances the learning experience by providing an interactively-rich environment where every student receives personalized tutoring. JITS is an online website always available for students and requires only a browser and an internet connection. JITS was designed to be accessible from remote locations and can be effectively used for distance education.

Conclusions

JECA demonstrates a Proof of Concept that can be effectively used to assist beginner Java programmers. JECA was originally implemented in JFlex and CUP, however, it became clear that JavaCC offered the greatest control and flexibility over error recovery and error correction; thus, JavaCC was used as the core lexical analyzer/parser generator tool.

JECA error corrects by intelligently learning and changing source program code and identifies errors clearly. The goals achieved by JECA include:

i) intelligently recognizing 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 s/he makes;

v) producing a 'modified code' that will compile (or bring the code closer to a state of successful compilation); and

vi) prompting the student programmer for more 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 research project, a Proof of Concept (i.e., JECA), was developed that fulfils the intended goals and assists the student in learning to better program in a more enjoyable way in the Java Intelligent Tutoring System.

JITS was designed and developed to provide rich interaction with students thus reducing the off-task time and student frustration. ITS researchers believe these are important issues when designing Intelligent Tutoring Systems (Anderson, 1998; Heffernan & Koedinger, 2001). Furthermore, the Java Intelligent Tutoring System was designed with efficient error remediation not to burden the student. JITS simply points

out an error without elaboration to a student when a mistake occurs. This is the recommended line of action as described in ACT-R theory (Anderson, 1998; Anderson et al., 1990).

JITS is being field-tested at the Sheridan Institute of Technology and Advanced Learning by students in the School of Applied Computing and Engineering Sciences, Ontario, Canada. It is hoped that the Java Intelligent Tutoring System will provide an interactively-rich learning environment for students that will result in increased achievement.

Recommendations

This research project focused on JECA and JITS. Although aspects of the field of Artificial Intelligence (AI) were included, more research on this area could raise some interesting findings. The following list presents some of the features that could be incorporated into JECA and JITS that implement AI strategies:

1. conduct the same activities that JECA performs but using AI techniques;
2. learn each student's unique learning style (e.g., visual vs. text based); and use this information to refine the tutoring process so that each student receives maximum benefit from JITS (see Figure 11);

3. determine traits among all (or many) of the students; this information could be used to address sections of specific types of problems or programming constructs that students find too difficult or too easy, etc.

Additional recommendations regarding JECA and JITS would be to conduct extensive field-tests with students and teachers to determine how the system could be improved. By encouraging numerous users with various levels of programming background to try out the system, information could be gathered and result in enhancing and optimizing specific aspects of JITS.

References

Aho, V. A., Sethi, R., & Ullman, D. J. (1988). *Compilers: Principles, Techniques, and Tools*. Menlo Park, CA: Addison-Wesley.

Aleven, V., & Ashley, D. K. (1997). Case-Based Argumentation Through a Model and Examples: Empirical Evaluation of an Intelligent Learning Environment. In B. Du Boulay & R. Mizoguchi (Eds.), *International Journal of Artificial Intelligence in Education* (pp. 87-94). Amsterdam: IOS Press.

Anderson, J. R. (1998). Production Systems and the ACT-R Theory. In P. Thagard (Ed.), *Mind readings: Introductory selections on cognitive science* (pp. 59-76). Cambridge, MA: MIT Press.

Anderson, J. R., Boyle, C. F., Corbett, A. T., & Lewis, M. W. (1990). Cognitive Modelling and Intelligent Tutoring. *Artificial Intelligence, 42*, 7-49.

Anderson, J. R., Conrad, F. G., & Corbett, A. T. (1989). Skill Acquisition and the Lisp Tutor. *Cognitive Science, 13*(4), 467-505.

Anderson, J. R., Corbett, A. T., Koedinger, K. R., & Pelletier, R. (1995). Cognitive Tutors: Lessons learned. *The Journal of the Learning Sciences, 4*, 167-207.

Anderson, J. R., & Pelletier, R. (1991). *A Development System for Model-Tracing Tutors.* Paper presented at the The Internation Conference on the Learning Sciences, Northwester University, Evanson, Illinois, USA.

Bennett, J. P. (1996). *Introduction to Compiling Techniques, A First Course using ANSI C, LEX, and YACC.* Berkshire, England: McGraw-Hill.

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

Chen, E. (2004). *Java: A False Sense of Security?* Retrieved Nov 10, 2004, from http://www.trendmicro.com/en/about/news/coverage/eva-chen.htm

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

Graesser, A. C., Person, N. K., & Harter, D. (2001). Teaching tactics and dialog in autotutor. *International Journal of Artificial Intelligence in Education, 12*, 12-23.

Heffernan, N. T., & Koedinger, K. R. (2001). *The Design and Formative Analysis of a Dialog-Based Tutor.* Paper presented at the AI in Education 2000 Workshop on Building Dialogue Systems.

Hudson, S. (1999). *CUP User Manual.* Retrieved September 29, 2004, 2004, from http://www.cs.princeton.edu/~appel/modern/java/CUP/

Klein, G. (2004). *JFlex User Manual.* Retrieved September 29, 2004, 2004, from http://www.jflex.de/

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

Norvell, T. S. (2004). *The JavaCC FAQ.* Retrieved September 29, 2004, 2004, from http://www.engr.mun.ca/~theo/JavaCC-FAQ

O'Reilly, R. C., & Munakata, Y. (2000). *Computational Explorations in Cognitive Neuroscience.* London, England: MIT Press.

Pawlan, M. (2004). *J2EE Tutorial.* Retrieved October 15, 2004, from http://java.sun.com/j2ee/1.3/docs/

Regian, F. D. (1997). Increased performance gains in individualized human tutoring. *IEEE: Intelligent Systems, 4*, 14-29.

Sreenivasa, V. (2004). *JavaCC User Manual.* Retrieved September 29, 2004, 2004, from https://javacc.dev.java.net/

Sykes, E. R. (2003). Java Intelligent Tutoring System Model and Architecture. *Proceedings of American Association of Artificial Intelligence Spring Symposium on Human Interaction with Autonomous Systems in Complex Environments*, 187-193.

Sykes, E. R., & Franek, F. (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*, 78-83.

Woolf, Beck, Eliot, & Stern. (2001). Growth and maturity of intelligent tutoring systems: A status report. In K. D. Forbus & P. J. Feltovich (Eds.), *Smart machines in education* (pp. 100-144). Cambridge, MA: MIT Press.

Woolf, B. P., Beck, J., Eliot, C., & Stern, M. (2001). Growth and maturity of intelligent tutoring systems: A status report. In K. D. Forbus & P. J. Feltovich (Eds.), *Smart machines in education* (pp. 100-144). Cambridge, MA: MIT Press.