PLAIN, AN OBJECT-ORIENTED MINI-LANGUAGE TO INTRODUCE
OBJECT-ORIENTED PROGRAMMING

GRADUATE PROJECT

BY

DAVID G. GWALTNEY

FEBRUARY 9, 2001

COMMITTEE MEMBERS

Dr. Holly Patterson-McNeill,
Chairperson

Dr. Michelle Moore,
Member

Dr. Stephen Dannelly,
Member

[Signatures]
ABSTRACT

This project is the design and implementation of a mini-language to introduce and simplify the practice of object-oriented programming. The Plain language is a subset of the Java language and is implemented using the Java-based lexical analyzer and parser generation tools JLex and CUP. Plain is a simple, easy-to-use programming language for first-time object-oriented programmers.
# TABLE OF CONTENTS

Abstract .................................................................................................................. i

Table of Contents ............................................................................................................. ii

List of Tables .................................................................................................................. iv

List of Figures .................................................................................................................. v

Introduction and Background .......................................................................................... 1

Narrative .................................................................................................................. 3

The Plain Language ...................................................................................................... 4

Plain Classes and Objects ............................................................................................... 4

Creating Plain Objects .................................................................................................... 5

Plain Data Types and Declarations ................................................................................ 6

Plain Methods ............................................................................................................... 7

Plain Statements .......................................................................................................... 8

Plain Expressions ......................................................................................................... 15

Inheritance .................................................................................................................. 20

Method Overloading and Overriding ............................................................................. 21

Plain Built-in Classes .................................................................................................... 23

Comments .................................................................................................................... 30

The Plain Interpreter ..................................................................................................... 31

Lexical Analysis ............................................................................................................ 31

Parsing and Syntactical Analysis .................................................................................... 33

Abstract Syntax Generation ......................................................................................... 34

Semantic Analysis ...................................................................................................... 36
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synthesis Into Java Source Code</td>
<td>38</td>
</tr>
<tr>
<td>Execution of Java Source Code</td>
<td>38</td>
</tr>
<tr>
<td>The Plain Graphical User Interface (GUI)</td>
<td>38</td>
</tr>
<tr>
<td>Environment</td>
<td>41</td>
</tr>
<tr>
<td>Results</td>
<td>42</td>
</tr>
<tr>
<td>System Design</td>
<td>45</td>
</tr>
<tr>
<td>Check Program Mode</td>
<td>48</td>
</tr>
<tr>
<td>Run Program Mode</td>
<td>49</td>
</tr>
<tr>
<td>Text Editor Mode</td>
<td>51</td>
</tr>
<tr>
<td>The Plain Interpreter</td>
<td>52</td>
</tr>
<tr>
<td>Lexical Analysis</td>
<td>54</td>
</tr>
<tr>
<td>Syntactical Analysis</td>
<td>58</td>
</tr>
<tr>
<td>Semantic Analysis</td>
<td>65</td>
</tr>
<tr>
<td>Conversion</td>
<td>70</td>
</tr>
<tr>
<td>Execution</td>
<td>71</td>
</tr>
<tr>
<td>Summary and Future Work</td>
<td>72</td>
</tr>
<tr>
<td>Bibliography</td>
<td>74</td>
</tr>
<tr>
<td>Appendix A. Plain Reserved Words</td>
<td>75</td>
</tr>
<tr>
<td>Appendix B. Plain BNF</td>
<td>76</td>
</tr>
<tr>
<td>Appendix C. Plain Syntax Charts</td>
<td>78</td>
</tr>
<tr>
<td>Appendix D. Source Code</td>
<td>84</td>
</tr>
</tbody>
</table>
LIST OF TABLES

Table 1. Plain Relational Operators ................................................................................ 17
Table 2. Plain Logical Operators ...................................................................................... 17
Table 3. Plain Arithmetic Operators ............................................................................... 18
Table 4. Operator Precedence ......................................................................................... 19
Table 5. Plain GUI Menu Options .................................................................................... 39
Table 6. Plain Class Variables ........................................................................................ 46
Table 7. Abstract Syntax Tree (AST) Classes ................................................................. 63
Table 8. Symbol Table Entry (STEntry) Classes ............................................................. 66
Table 9. TypeCheck Class Methods ............................................................................... 69
Table 10. Converter Class Methods ............................................................................... 70
LIST OF FIGURES

Figure 1. Plain Interpreter Implementation ................................................................. 31
Figure 2. Symbol Table Implementation ................................................................. 37
Figure 3. Plain GUI Screen Capture ....................................................................... 40
Figure 4. Plain Programming Environment Level 1 DFD ................................. 45
Figure 5. Plain GUI Level 2 DFD (1.1) ................................................................. 46
Figure 6. Plain GUI State Transition Diagram ...................................................... 51
Figure 7. Plain Interpreter Level 2 DFD (1.2) ....................................................... 53
Figure 8. Plain Programming Environment Structure Chart .............................. 53
INTRODUCTION AND BACKGROUND

The popularity of Object-Oriented Programming (OOP) has increased dramatically during recent years. The benefits of abstraction, inheritance and encapsulation have increased the efficiency of the programming task while simultaneously improving the maintainability and reusability of the programs produced. The object-oriented age, however, has not yet come to full fruition. Most object-oriented languages available today require some degree of programming experience for their benefits to be fully realized. While Java has improved understandability over C++, it is still somewhat imposing for beginning OOP programmers to achieve all except the most basic of tasks. A great deal of understanding is required to produce useful programs.

A simple, easy to understand OOP mini-language will bridge the gap between beginning and experienced programmers. It will allow the introduction of OOP concepts and practices in a controlled, limited environment, allowing full understanding prior to exposure to large, cumbersome OOP languages. Numerous mini-languages have been developed to exploit the intricacies of procedural languages, yet OOP programmers and teachers of them are forced to spend weeks just preparing to write a useful program.

At the other end of the programming spectrum, compiler implementation has rarely been attempted using OOP languages. Until recently, lex and yacc were largely the only compiler development tools used (or available) to teach compiler design. Many students of computer science today have little to no experience in procedural languages, and must take a step backward to fully realize the complexity of compiler design. It is time to take OOP technology full-circle, and the tools are available to make it happen. Object-
oriented technology is the future and, therefore should be fully implemented from introductory through advanced programming topics.
NARRATIVE

The OOP mini-language project involved the design of the Plain mini-language and the implementation of an interpreter to convert source code in that language into Java source code. The Plain language design includes both formal and informal language descriptions to enable inclusion in an educational environment. The interpreter implementation also includes full documentation to serve as a basis for inclusion in future compiler design courses.

Plain provides a graceful introduction to OOP concepts and programming by including OOP terminology as part of the language itself. When defining a subclass for example, the reserved word “inherits” is used instead of “extends” to enhance programmer understanding of inheritance. Similar conventions are used whenever possible throughout the language. Minimizing the data types to essential types including integer, String, input/output and various graphics required to build minimal Graphical User Interface (GUI) applications have reduced complexity. Reduction of program control-structure constructs and variance within them enables a set of straightforward programming tools that are easy to grasp and implement while maintaining the flexibility required for efficient programming. The mini-language user interface provides a Windows environment for entering source code, with drop-down menus for compilation and execution of the user programs.

The implementation of the interpreter is in Java, facilitating an introduction to the processes and practices involved in OO compiler construction as opposed to procedural language implementations. The Java-based lexical-analyzer generator JLex was used to provide lexical analysis and the Java-based Constructor of Useful Parsers (CUP) was
used to generate a parser for the language and perform syntactical analysis. Semantic analysis is performed by two additional Java modules. User-friendly and informative error messages are generated to assist the programmer in correcting syntax and semantic errors. The interpreter accepts the source code of the mini-language and converts it into Java source code for execution in the Plain programming environment.

**The Plain Language**

A Plain program consists of one or more classes. Classes define objects, which model the actual or logical entities involved with the problem to be solved. Once a class is defined, an object belonging to that class can be created and used in a program. A class contains variables and methods. Variables contain the data that define the “state” of the object, while methods are the means by which those variables are manipulated and actual program operations are performed.

**Plain Classes and Objects**

A Plain class definition follows the form:

```
objectClass  identifier  [inherits  identifier]
  {{[VarDecls][MethodDecls]}}
```

where *identifier* is a valid Plain identifier, *VarDecls* is zero or more variable declarations and *MethodDecls* is zero or more method declarations. The optional *inherits* clause allows a newly declared class to inherit attributes from previously declared classes. This concept is discussed further in the *Inheritance* section of this narrative.

Plain classes, variables and methods are given user-assigned names called identifiers, with the exception of the *main* method. Plain identifier names can be from 1 to 31
characters in length. Identifier names must start with a letter or an underscore, and can be followed by any combination of letters, numbers or underscores. Plain is case sensitive, hence count, Count and COUNT are three separate identifiers. A Plain identifier cannot be the same as a Plain reserved word.

Once a class is defined, the identifier becomes a new data type and can be used to declare and create objects of that type. The following Plain program creates a simple class named Vehicle:

```plain
objectClass Vehicle{
    object make := new String("");
    object model := new String("");
}
```

In the program above and all future examples, Plain reserved words are shown in bold. A complete listing of Plain reserved words is located in Appendix A.

Creating Plain Objects

Plain objects are created through special methods called constructors. Constructors perform initialization when an object is created and must have the same name as the class itself. The general form of a Plain constructor is:

```plaintext
method construct identifier(formal-argument-list) {
    statement-list
}
```

where identifier is the type of object to be created (the class of the object), formal-argument-list contains the formal parameters for initialization, and statement-list is one or more Plain statements.

The following Plain program illustrates implementation of a constructor with the Vehicle class created earlier.
objectClass Vehicle{
    object make := new String("");
    object model := new String("");
    method construct Vehicle(String myMake, String myModel){
        make := myMake;
        model := myModel; }
    method main(){
        object mainWindow := new MainWindow("My Window");
        object outBox := new OutputBox(mainWindow,"My Output");
        object myTruck := new Vehicle("Chevy","Z-71");
        mainWindow.show();
        outBox.show();
        outBox.printLine(myTruck.make + " " + myTruck.model);}
}

The constructor for the Vehicle class contains two String parameters—myMake and myModel. The constructor method is invoked by the assignment statement:

    object myTruck := new Vehicle("Chevy","Z-71");

which creates a new Vehicle object named myTruck through use of the Plain reserved word new. The actual parameters “Chevy” and “Z-71” are assigned to the constructor formal parameters myMake and myModel respectively. Within the constructor, these values are assigned to the class variables make and model. The variables for the new Vehicle object myTruck are then accessed through member access (or “dot”) notation for output (myTruck.make and myTruck.model). The above program consists of only one method besides the constructor—main. The main method is a special Plain method that is required to be included in one of the classes in each Plain program. The main method is the first method executed when you run a program.

Plain Data Types and Declarations

A variable is a named location in memory that is used to hold a value that may be modified by the program. All Plain variables must be declared before they can be used. The general form of a variable declaration is:
variable_type identifier [:= expression];

where variable_type is a valid Plain or user-defined data type, identifier is a valid Plain identifier, and expression is a valid Plain expression setting the initial value of the newly created variable. There are two types of variables in Plain: class and method. Class variables are declared within the class definition, but outside of any method. Class variables can only be accessed by objects defined by that class (or a class which inherits from that class). Method variables are declared within the method to which they pertain. Method variables can only be accessed by the method which contains them.

The Plain language supports integer, boolean and object data types. Integer variables can hold a number in the range –2,147,483,648 to 2,147,483,647 and are declared using the integer reserved word. Boolean variables can hold true or false and are declared using the boolean reserved word. Object data types consist of user-defined class objects and Strings and are declared using the object reserved word.

A Plain String is a sequence of characters. Strings were used in the previous two sample programs to hold make and model data. Two Strings can be concatenated using the “+” operator as in the following statement from the Vehicle example:

```plaintext
outBox.printLine(myTruck.make + " " + myTruck.model);
```

This statement causes a space (“ ”) and the String contents of myTruck.model to be concatenated to the String myTruck.make.

**Plain Methods**

The general form of a Plain method declaration is:

```plaintext
method [return-type] identifier([parameter-list]) {
    statement-list
}
```
where \textit{return-type} is optional and defines the Plain data type returned by the method, \textit{identifier} is a valid Plain identifier and \textit{parameter-list} contains the formal parameters which accept the values (or a reference to the values) being passed to the method. Integer and boolean variables are passed by value, while objects (including Strings) are passed by reference. This means that unless an integer or boolean variable passed to the method is also returned by the method, any modification will not be reflected in the rest of the Plain program. The Plain Statements section describes how methods are invoked.

\textbf{Plain Statements}

Plain statements are the tools with which Plain methods are built. Statements are organized in a linear fashion to accomplish the tasks associated with each method. Plain consists of five basic types of statements: declaration, assignment, selection, iteration and method control. Declaration statements are used to declare variables within a method and follow the variable declaration format described above. The Plain assignment statement is composed using the assignment operator (\texttt{:=}). The Plain selection statement is the \texttt{if} statement. Plain iteration statements are the \texttt{for} statement and the \texttt{while} statement.

There are two types of method control statements in Plain: method call and method return statements. Method calls are used to transfer program control from the currently executing method to another program method, while a method return statement is used to return a value and program control to the calling method. All Plain declaration, assignment and method control statements are terminated with a semicolon (\texttt{;}). The following paragraphs contain more detailed information on Plain statements.
Assignment Statement

The Plain assignment statement consists of a variable, the assignment operator (:=), and an expression. The general form of the Plain assignment statement is:

\[ \text{identifier} := \text{expression}; \]

where \text{identifier} is a declared Plain variable identifier, and \text{expression} is a valid Plain expression. The Plain assignment statement causes the value of the expression on the right-hand side of the assignment operator to be assigned to the identifier on the left-hand side of the assignment operator. The expression on the right-hand side must match the type of the identifier on the left-hand side. The code segment below illustrates the three valid uses of the assignment statement, and would cause “the numbers are 10, 20 and 30” to be displayed.

```plaintext
objectClass some-numbers{

    method main() {
        object mainWindow := new MainWindow("My Window");
        object outBox := new OutputBox(mainWindow,"My Output");
        integer num1;
        integer num2;
        integer num3;

        num1 := 10;
        num2 := num1 * 2;
        num3 := someNumbers.get_a_number(num2);
        mainWindow.show();
        outBox.show();
        outBox.printLine("the numbers are " + num1 + " " + num2 + " " + num3);
    }

    method integer get_a_number(integer numf) {
        integer temp;
        temp := numf + 10;
        return temp;
    }
}
```

The following sequence of events causes the given results:

1) \text{num1} is directly assigned the integer value 10.
2) num2 is assigned the value of num1 multiplied by 2.

3) num3 is assigned the value which results from invoking the method

get_a_number as follows:

   a. The value of num2 (20) is passed to the method get_a_number.

      (get_a_number(num2))

   b. The value is of num2 is received and assigned to numf.

   c. The variable temp is assigned the value of numf plus 10 (30).

   d. The value of temp is returned and assigned to num3.

if Statement

The Plain if statement is a selector construct which selects between two courses of
program flow, dependent on the result of a conditional expression. If the conditional
expression evaluates to true, the then clause is executed, otherwise the else clause—
which is optional—is executed. The general form of the if statement is:

   if (expression) then {statement-list}
   [else {statement-list}]

where expression is a Plain expression which evaluates to true or false, and

statement-list is a Plain statement or statements. If the conditional expression
evaluates to true, the then clause (the statement(s) following the then) is executed;
otherwise, the else clause is executed. The else clause is optional.

The following code-segment example illustrates the use of the Plain if statement:

    object mainWindow := new MainWindow(“My Program”);
    object inBox := new InputBox(mainWindow, “Numbers”,
                               “Entry must be an integer”)
    object outBox := new OutputBox(mainWindow,”Results”)
    integer num1;
    integer num2;
    integer num3;
mainWindow.show();
num1 := inBox.getInteger(“Enter number”);
num2 := num1 * 3;
num3 := num1 + 3;
outBox.show();
if (num1 < 2) then
{
    outbox.printLine(num3 + “ is greater than ” + num2);
}
else {
    outBox.printLine(num2 + “ is greater than ” + num3);
}

In the code segment above, if the user input accepted (via inBox.getInteger() method) into the variable num1 is 1, num3 will be greater than num2, and the output of this code segment will be: 4 is greater than 3. The else clause will not be executed, and program execution will continue with the first statement after the else clause. If the number entered is 2, the then clause would be skipped, and the else clause statement would render the output: 6 is greater than 5.

Plain if statements can also be nested within each other. The following example illustrates nested if statements:

num1 := inBox.getInteger(“Enter number”);
num2 := num1 * 2;
num3 := num1 + 2;
outBox.show();
if (num1 > 1) then  /*outer if*/
{
    if (num1 = 2) then  /*inner if*/
    {
        outBox.printLine(num2 + “ is equal to ” + num3);
    }
    else  /*inner else * /
    {
        outBox.printLine(num2 + “ is greater than ” + num3);
    }
}
else  /*outer else*/
In this case, if the number entered is greater than 1, control is passed to the *inner* if statement to determine if *num1* is *equal* to 2. If it is, the *inner* if would produce the appropriate output; otherwise the *inner* else clause would produce the output and program control would resume after the *outer* else clause.

If the number entered is not greater than 1, the *inner* if statement (including the *inner* else clause) would be skipped and the *outer* else clause would be executed.

*for* Statement

The Plain *for* statement is an iteration construct that provides a means for executing a loop of program code multiple times. The loop is controlled through manipulation of a *loop control variable*. The *for* statement allows for *initialization*, *comparison* via an expression, and *increment* operations using the loop control variable. Until the conditional expression evaluates to *false*, the iteration will continue. The general form of the *for* statement is:

```
for(init-statement; expression; inc-statement;)
{statement-list}
```

where *init-statement* is an assignment statement which initializes the control variable to a specific value, *expression* is a comparison of the control variable to some threshold value, *inc-statement* is an assignment statement which increases the value of the control variable by one, and *statement-list* is a Plain statement or statements
enclosed in brackets ({}), which is to be executed until the conditional expression is no longer true. The control variable must be declared prior to use within the for statement.

The following example illustrates implementation of a Plain for statement. The loop control variable is i.

```plaintext
for(i := 1; i < 10; i := i + 1)
{num1 := num1 + 5;}
```

The for statement above would execute 10 times before exiting as follows:

1) The variable i is initialized to a value of 1 (initialization).
2) The conditional expression (i < 10) is evaluated (comparison).
3) The variable i is incremented by 1 (increment).
4) The statement (num1 := num1 + 5) is executed.
5) Steps 2 through 4 are repeated until i is equal to 10.

As the following example illustrates, it is also possible to nest Plain for statements.

```plaintext
for(i := 1; i < 10; i := i + 1)        /*outer for*/
{
  for(j := 1; j < 10; j := j + 1)   /*inner for*/
  {num1 := num1 + 5;}
}
```

In the case of nested for statements, control will be passed to the inner for until the outer for conditional expression (i < 10) evaluates to false. In each instance, the inner for will be executed until the inner conditional expression (j < 10) evaluates to false. This means that while the outer for will be executed 10 times, the inner for will execute 100 times: 10 times for each time the outer for is executed.
while Statement

The Plain while statement is an iteration construct that allows execution of a block of program statements based solely on a condition. While the for statement is useful for a predetermined amount of iterations, the while statement is well suited for situations in which the number of iterations is unknown. The general form of the while statement is:

```
while (expression) {statement-list}
```

where expression is a Plain expression which evaluates to true or false, and statement-list is a Plain statement or statements enclosed in brackets ([]) to be executed until the conditional expression is no longer true. The following code segment illustrates implementation of the Plain while statement:

```
while (num1 < num2) {
    num1 := num1 + 1;
}
```

In the example above, the statement \texttt{num1 := num1 + 1} will be executed continuously until the conditional expression \(\texttt{(num1 < num2)}\) evaluates to \textit{false}.

As with all Plain control statements, it is possible to nest while statements. The following example illustrates nested while statements:

```
while (num1 < num2) /*outer while*/ {
    num1 := num1 + 1;
    while (num1 < num3) /*inner while*/ {
        num3 := num3 - 1;
    }
}
```

The \textit{inner} while in the example will only execute until \texttt{num3} has been reduced to a value less than \texttt{num1}. The \textit{outer} while will execute until \texttt{num1} is greater than \texttt{num2}. If
num1 is greater than num2 initially, the inner while will never execute. After completion, program control will transfer to the first statement following the closing brace of the outer while.

*Method Control Statements*

Plain method control statements have two distinct forms: method call and method return. A method call is a Plain statement designed to simply transfer control of the program to the indicated method. Method call statements pass control to methods that do not return a value. The general form of the method call statement is:

```
[expression.]identifier([parameter-list]);
```

where *identifier* is the name of the method to be invoked, *parameter-list* contains the actual parameters to pass to the method, and *expression* resolves to the class which contains the method being called.

The Plain method return statement is used to return control to the calling method once an invoked method has finished executing. It has two forms: one which returns a value and one which doesn’t. The following summarizes the two forms:

```
return;
```

or

```
return expression;
```

where *expression* is the method identifier containing the value to be returned or a valid expression or value. The return type must match the method type.

*Plain Expressions*

Plain expressions are an integral part of the Plain language. As reflected in the previous section on Plain statements, expressions are the building blocks for most statements.

Conditional expressions are used in all of the Plain selection and iteration statements to
control program behavior. Conditional expressions are those which evaluate to true or false. Arithmetic expressions are used in Plain assignment statements to change the value of variables. Arithmetic expressions are those which evaluate to an integer value. Plain methods that return values are also used as expressions in assignment statements. These are the method call expression and the `new` expression.

**Conditional Expressions**

Plain conditional expressions evaluate to true or false. The general form of a conditional expression is

\[(\text{expression1 relational-operator expression2})\]

where `expression1` and `expression2` are valid identifiers or values and `relational-operator` is a valid Plain relational operator. Plain expressions can also be compound in nature. That is, more than one conditional expression can be evaluated at a time. Multiple conditional expressions are evaluated using Plain logical operators. The general form of a compound Plain conditional expression is

\[
((\text{expression1 relational-operator expression2})
\text{ logical-operator}
(\text{expression3 relational-operator expression4}))
\]

where `expression1` through `expression4` are valid identifiers or values, `relational-operator` is a valid Plain relational operator, and `logical-operator` is a valid Plain logical operator.

**Relational Operators**

In the term `relational operator`, relational refers to the relationships that values can have with one another. For example, a value can be equal to another, greater than another, or
less than another. Relational operators are used to evaluate these relationships. Table 1 outlines the Plain relational operators.

**Table 1. Plain Relational Operators**

<table>
<thead>
<tr>
<th>Relational Operator</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Equal to</td>
<td><code>if(num1 = num2) then...</code></td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
<td><code>while (num1 &gt; num2)...</code></td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
<td><code>if (num1 &lt; num2)...</code></td>
</tr>
</tbody>
</table>

**Logical Operators**

In the term *logical operator*, logical refers to the way that relationships can be connected. In Plain, logical operators are used to create compound conditional expressions. Table 2 lists the two Plain logical operators and their meaning.

**Table 2. Plain Logical Operators.**

<table>
<thead>
<tr>
<th>Logical Operator</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>OR</td>
<td><code>if ((num1 = num2) OR (num3 &gt; num4)) then...</code></td>
</tr>
<tr>
<td>AND</td>
<td><code>while ((num1 = num2) AND (num3 &lt; num4))...</code></td>
</tr>
</tbody>
</table>

In the case of the **AND** operator example, the statement(s) of the then clause would be executed only if both conditional expressions evaluated to true (only if `num1` equals `num2` AND `num3` is greater than `num4`). In the **OR** example, the statement(s) following the **while** would execute if **either** conditional expression evaluated to true (if `num1` equals `num2` OR `num3` is less than `num4`).

**Arithmetic Expressions**

Plain arithmetic expressions are used to assign integer values to Plain variables through the use of the assignment operator. The general form of a Plain arithmetic expression is:

\[ \text{expression1 arithmetic-operator expression2} \]
where expression1 and expression2 are integer variables, integer literals or Plain arithmetic expressions, and arithmetic-operator is a valid Plain arithmetic operator.

As the definition above explains, compound arithmetic expressions are possible so long as each arithmetic operator involves two expressions that evaluate to integers.

Parenthesis may be necessary to ensure that compound arithmetic expressions are evaluated in the proper sequence.

**Arithmetic Operators**

Plain provides five arithmetic operators for use in developing programs. The Plain arithmetic operators are listed and explained in the following table.

<table>
<thead>
<tr>
<th>Arithmetic Operator</th>
<th>Operation</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
<td>A + B</td>
<td>Sum of A added to B</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
<td>A - B</td>
<td>Difference of A minus B</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>A * B</td>
<td>Product A multiplied by B</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td>A / B</td>
<td>Truncated quotient of A divided by B</td>
</tr>
<tr>
<td>%</td>
<td>Modulus division</td>
<td>A % B</td>
<td>Remainder of A divided by B</td>
</tr>
</tbody>
</table>

NOTE: Although the examples in the table use variables A and B, integer values may also be used.

**Operator Precedence**

In order to facilitate the creation of the compound expressions discussed in the paragraphs above, precedence must be established for the logical, relational and arithmetic operators used in Plain programs. Without a precedence for Plain operators, the programmer would be forced to conform to some unnatural rules—strictly left to right, for example—when composing expressions. The Plain precedence for operators follows those of most other programming languages and also obeys the natural
mathematics rules that most of us have become familiar with. The following table summarizes operator precedence in Plain.

Table 4. Plain Operator Precedence.

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operator(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Highest</td>
<td>* / %</td>
</tr>
<tr>
<td></td>
<td>+ -</td>
</tr>
<tr>
<td>Lowest</td>
<td>AND</td>
</tr>
<tr>
<td></td>
<td>OR</td>
</tr>
<tr>
<td></td>
<td>= &lt; &gt;</td>
</tr>
</tbody>
</table>

NOTE: In the case of multiple operators with the same precedence, expressions will be evaluated from left to right.

You can add tabs and spaces to Plain expressions to make them easier to read.

Additionally, redundant or additional parentheses do not cause errors in Plain.

Expressions enclosed in parenthesis are evaluated first. In the case of nested expressions, the innermost expression is evaluated first.

**New Expression**

The Plain `new` expression is used to create a new object of the specified type by invoking a constructor method for the object type to be created. The general form of the `new` expression is:

```
new type([parameter-list])
```

where `type` is a valid object type (String or user-defined) and `parameter-list` consists of the actual parameters required by the constructor method.

**Method Call Expression**

The Plain method call expression is used as the right hand side of an assignment statement to assign method call results to an identifier. The general form of the assignment method call expression is:

```
[expression.]identifier([parameter-list]);
```
where *identifier* is the name of the method to be invoked, *parameter-list* contains the actual parameters to pass to the method, and *expression* resolves to the class which contains the method being called.

**Inheritance**

The Plain language supports inheritance, a mechanism to define new objects based on existing ones by creating a new *objectClass* which inherits all the members from the *objectClass* from which it is derived. The new *objectClass* is called a subclass, and the existing class is the superclass. Methods and variables defined in the superclass are automatically available to the subclass, and additional members may be added.

Implementation is through use of the inherits clause in the class definition. The following Plain program illustrates implementation of inheritance using the *Vehicle* class defined earlier.

```plaintext
objectClass Vehicle{
    object make := new String(""),
    object model := new String(""),
    method construct Vehicle(String myMake, String myModel)
    {
        make := myMake;
        model := myModel;
    }
}

objectClass PickUpTruck inherits Vehicle{
    integer loadRating;
    method construct PickUpTruck(String mfr, String mdl,
        integer rtg)
    {
        super(mfr,mdl);
        loadRating := rtg;
    }
    method main()
    {
        object mainWindow := new MainWindow("My Window");
        object outBox := new OutputBox(mainWindow,"My Output");
        object myTruck := new PickUpTruck("Chevy","Z-71",1000);
    }
}
```
The newly defined `PickUpTruck` class adds a variable `rtg` to record the load rating of the truck. The constructor for the `PickUpTruck` initializes the new object by first calling the superclass constructor using the reserved word `super`, which invokes the constructor for the superclass and initializes the make and model variables of `myTruck`. The additional `loadRating` variable is then initialized, and the resulting output is:

```
Chevy Z-71 1000 lbs
```

Method Overloading and Overriding

Plain supports method overloading, which allows the design of multiple versions of a method, each with a unique signature. A method’s signature is a combination of its name and the number, order and type of arguments. The following code segment defines a modified Pick-Up-Truck class and illustrates method overloading:

```
objectClass Pick-Up-Truck inherits Vehicle{
    integer loadRating;
    integer odometer;
    method construct Pick-Up-Truck(String mfr, String mdl,
    integer rtg, integer miles)
    {
        super(mfr, mdl);
        loadRating := rtg;
        odometer := miles;
    }
    method main()
    {
        integer myMiles;
        object mainWindow := new MainWindow("My Window");
        object outBox := new OutputBox(mainWindow,"My Output");
        object myTruck := new Pick-Up-Truck
        ("Chevy","Z-71",1000,12000);
        myMiles := myTruck.mileage();
        mainWindow.show();
    }
}
```
```java
outBox.show();
outBox.println(myMiles + " miles");
myTruck.mileage(12530);
myMiles = myTruck.mileage();
outBox.println(myMiles + " miles");
}
method mileage(integer truckMiles)
{
    odometer = truckMiles;
    return;
}
method integer mileage()
{
    return(odometer);
}
}

Although the return types are different in the two mileage methods, the signature difference is the determining factor in which method is invoked. Had the signatures been the same, an error would have occurred. The end result of the class above is the creation of two mileage methods—one which sets the mileage, and one which retrieves it.
Method overloading occurs when the same method name is used within the same class. Plain also supports method overriding, which occurs when a subclass method name also exists in its superclass. The subclass method will then override the superclass method. With method overriding, the return type and signatures of the two methods must be exactly the same. The following example illustrates method overriding:

```objectClass`` Vehicle{
    object make = new String(""");
    object model = new String("");
    integer odometer;
    method construct Vehicle(String mfr, String mdl, integer miles)
    {
        make = mfr;
        model = mdl;
        odometer = miles;
    }
    method mileage(integer vehicleMiles)
    {
```
objectClass OldTruck inherits Vehicle{
  method construct Old-Truck(String mfr, String mdl, Integer odoReading)
  {
    super(mfr,mdl,odoReading);
  }
  method mileage(integer truckMiles)
  {
    if(truckMiles > 100000) then
    {
      odometer := 99999;
    }
    else
    {
      super.mileage(truckMiles);
    }
    return;
  }
  method main()
  {
    integer myMiles;
    object myTruck := new OldTruck
      ("Chevy ","C-10 ",120000);
    myTruck.mileage(125030);
  }
}

In this example, the subclass (OldTruck) mileage method overrides the superclass (Truck) mileage method since the old truck odometer is limited to displaying mileage below 100,000 miles.

Plain Built-in Classes

Plain provides several classes to assist in the rapid creation of working programs. The availability of these classes allows the programmer to concentrate on the implementation details of the programming problem instead of the details of utility. The following paragraphs describe the Plain built-in classes.

---

1 The Plain built-in classes are all freeware and part of the JavaBook Package created by C.Thomas Wu and downloaded from the Dr Caffeine web site. Some functionality has been omitted to simplify the classes.
MainWindow

The MainWindow class is used as the top-level window of a Plain application and is required to use several of the other built-in classes. The MainWindow window will be almost as big as the screen and positioned at the center of the screen. When the user closes this window, the program is terminated.

The MainWindow constructor method is

```java
method construct MainWindow(String title)
```

where title is the title of the window.

The MainWindow class has only one other method—the show method. This method makes the application window visible to the user and requires no arguments. The following is a small example of implementation of the MainWindow class:

```java
objectClass MyFirstApplication{
    method main()
    {
        object mainWindow := new MainWindow(); //create and
        mainWindow.show(); //display a window
    }
}
```

InputBox

The InputBox class is used for getting an input from the user. An InputBox is capable of accepting integers and Strings and will remain on the screen until a valid input is entered. This class requires a MainWindow for implementation and its constructor is:

```java
method construct InputBox(MainWindow owner, String title, String errorMsg)
```

where owner is an existing MainWindow object, title is the title of the InputBox window and errorMsg is the message to be displayed when the user enters invalid data.
The **InputBox** class has two associated methods—**getInteger** and **getString**—to obtain applicable user input and operate as follows:

**method integer getInteger(String text)**

A call to **getInteger** displays an **InputBox** and returns an integer value entered by the user. The user is prompted for input using the *text* parameter as the prompt text.

**method String getString(String text)**

A call to **getString** displays an **InputBox** and returns a String entered by the user. The user is prompted for input using the *text* parameter as the prompt text.

The following example method illustrates implementation of an **InputBox**:

```java
objectClass MyInputBox{
  method main()
  {
    integer myNumber;
    object mainWindow := new MainWindow("InputBox Demo");
    object myString := new String(""");
    object inputBox1 := new InputBox(mainWindow, "Number Entry", "Integer Required");
    object outBox := new OutputBox(mainWindow,"Program Output");
    object inputBox2 := new InputBox(mainWindow, "String Entry", "String Required");
    mainWindow.show();
    myNumber := inputBox1.getInteger("Enter number");
    myString := inputBox2.getString("Enter String");
    outBox.show();
    outBox.printLine("The number was: " + myNumber);
    outBox.printLine("String was: " + myString);
  }
}
```

**OutputBox**

The **OutputBox** class is used to display program text output. It requires a **MainWindow** for implementation and its constructor is:
method construct OutputBox(MainWindow owner, String title)

where owner is an existing MainWindow object and title is the title of the InputBox window. The OutputBox class has three associated methods as follows:

method printLine(String text)

where text is the String to be displayed in the OutputBox. This method prints out the text and moves the cursor to the next line.

method waitUntilClose()

This method causes program execution to pause until the OutputBox window is closed.

method show()

This method causes the OutputBox to be displayed.

The following example illustrates usage of the OutputBox class.

objectClass MyFirstOutputBox{
method main()
{
    object mainWindow := new MainWindow("My First Output Box");
    mainWindow.show();
    object outBox := new OutputBox(mainWindow,"My Output");
    outBox.show();
    outBox.printLine("This is an OutputBox");
}
}

ResponseBox

The ResponseBox class is used for accepting a YES/NO response from the user through use of two buttons. It requires a MainWindow for implementation and its constructor is:

method construct ResponseBox(MainWindow owner)
where owner is an existing MainWindow object.

The ResponseBox class has only one method—prompt—which displays the ResponseBox with a user-defined message:

```
method integer prompt(String text)
```

where text is the message to be displayed.

The ResponseBox class has two predefined class variables—YES and NO—for determining the user response. Both are integers and their use is illustrated in the following example:

```
objectClass ResponseBoxDemo{
  method main()
  {
    integer answer;
    object mainWindow := new MainWindow("Response Box Demo");
    object responseBox := new ResponseBox(mainWindow);
    object outBox := new OutputBox(mainWindow,"Program Output");
    mainWindow.show();
    answer := responseBox.prompt("Isn't programming fun?");
    if(answer = 1) then
      {
        outBox.show();
        outBox.println("You answered YES");
      }
    else
    {
      outBox.show();
      outBox.println("You answered NO");
    }
  }
}
```

**DrawingBoard**

The DrawingBoard class allows the drawing of lines, circles, and rectangles within a DrawingBoard window. The coordinate system used for all DrawingBoard methods is in pixels referenced from the upper left corner of the window (e.g., an [x,y] coordinate
of [0,0] represents the upper left corner of the DrawingBoard window). The

DrawingBoard constructor is defined as follows:

    method construct DrawingBoard(String title)

where title is the desired title of the DrawingBoard window.

The DrawingBoard class has four methods described as follows:

    method drawCircle(integer x, integer y, integer radius)

where the pair [x,y] is the coordinate of the center of the circle and radius is the radius of the circle in pixels.

    method drawLine(integer x1, integer y1, integer x2, integer y2)

where the coordinate [x1,y1] is the beginning of the line and coordinate [x2,y2] is the end of the line.

    method drawRectangle(integer x, integer y, integer width, integer height)

where the coordinate [x,y] represents the top, leftmost corner of the rectangle, width is the width of the rectangle and height is the height of the rectangle.

    method setVisible(boolean view)

where view determines whether or not the DrawingBoard is visible (true) or hidden (false).

The following example illustrates implementation of the DrawingBoard class:

    objectClass MyFirstDrawingBoard{
    method main()
    {
        object drawingBoard := new DrawingBoard("Drawing Board");
The SketchPad class supports a freehand drawing. You draw pictures by dragging the mouse while holding the left button down. You erase the drawing by clicking the right button. The constructor is:

```java
method construct SketchPad(String title)
```

where `title` is the title of the SketchPad window.

The SketchPad class has only one other method—the show method. This method makes the window visible to the user and requires no arguments. The following is a simple example implementation of SketchPad:

```java
objectClass FunTime{
    method main (){
        object doodleBoard := new SketchPad("DOODLE");
        doodleBoard.show();
    }
}
```

The Clock class supports two basic clock functions—reading the current time and getting today's date. The Clock constructor is

```java
method construct Clock()
```

The Clock class has two associated methods as follows:

```java
method String getCurrentDate()
```
This method returns a String representation of the current (today's) date in mm/dd/yyyy format.

```java
method String getCurrentTime()
```

This method returns the current time in 12-hour format.

The following example illustrates implementation of the Clock class:

```java
objectClass MySecondOutputBox{
    method main()
    {
        object mainWindow := new MainWindow("Main Window");
        mainWindow.show();
        object outBox := new OutputBox(mainWindow,"My Output");
        outBox.show();
        object myClock := new Clock();
        object date:=new String("");
        object time := new String("");
        date := myClock.getCurrentDate();
        time := myClock.getCurrentTime();
        outBox.printLine("Date is: " + date);
        outBox.printLine("Time is: " + time);
    }
}
```

**Comments**

In Plain, all multi-line comments begin with the character pair /* and end with */.

There must be no spaces between the asterisk and the slash. The compiler ignores any text between the beginning and ending comment symbols. Single line comments begin with the double slash (//) and continue to the end of the line. Comments may be placed anywhere in a program, as long as they do not appear in the middle of a keyword or identifier.
The Plain Interpreter

The Plain interpreter accepts user code entered or opened through the user interface window, performs lexical, syntactical and semantic analysis of the Plain source code, and converts the Plain source code into Java source code for compilation and execution within the Plain run-time environment. The phases of the implementation are depicted in Figure 1 and described in the following paragraphs.

Lexical Analysis

Lexical analysis was accomplished through use of the JLex lexical analyzer generator. JLex produces a Java program from a lexical specification of regular expressions. Like its predecessor lex, a JLex input file is organized into three sections, separated by double-percent directives (``%%''). A proper JLex specification has the following format:

```
user code
%%
JLex directives
%%
regular expression rules
```
The user code section allows users to write Java code for use by the lexical analyzer. The code included in this section is optional and it must be situated before the first `%%` delimiter. This code is copied directly at the beginning of the generated file. This section is mainly used to import packages or to define classes, variables and return types. In the JLex directives section, the lexical analyzer directives are written. Each directive must be located at the beginning of a single line.

In the third part of the JLex specification, rules are defined for breaking the input stream into tokens. The rules specify regular expressions and associate them with actions consisting of Java source code. The rules have three distinct parts: the optional state list, the regular expression, and the associated action.

The following is a small-example JLex specification presented in (Horwitz, 2000):

```plaintext

%%

DIGIT= [0-9]
LETTER= [a-zA-Z]
WHITESPACE= [ \	\n]      // space, tab, newline

// The next 3 lines are included so that we can use the generated scanner
// with java CUP (the Java parser generator)
%implements java_cup.runtime.Scanner
%function next_token
%type java_cup.runtime.Symbol

// Turn on line counting
%line

%%

(LETTER)((LETTER)|(DIGIT)*) {System.out.println(yyline+1 + ": ID " + yytext());}
{DIGIT}+                    {System.out.println(yyline+1 + ": INT");}
"="                      {System.out.println(yyline+1 + ": ASSIGN");}
"=="                     {System.out.println(yyline+1 + ": EQUALS");}
{WHITESPACE}*               { }
.                    {System.out.println(yyline+1 + ": bad char");}
```

32
Parsing and Syntactical Analysis

Parsing was accomplished through use of the Java-based Constructor of Useful Parsers (CUP). CUP is a system for generating Look-Ahead Left-to-right parser, Rightmost derivation (LALR) parsers from simple specifications. It is written in Java, uses specifications including embedded Java code, and produces parsers which are implemented in Java. The input to CUP is a specification that includes:

1) optional package and import declarations,
2) optional user code ,
3) terminal and nonterminal declarations ,
4) optional precedence and associativity declarations,
5) grammar rules with associated actions.

The following is a small-example CUP specification included with the CUP program documentation:

```java
import java_cup.runtime.*;

parser code {:
    public static void main(String args[]) throws Exception {
        new parser(new Yylex(System.in)).parse();
    }
};
terminal SEMI, PLUS, TIMES, LPAREN, RPAREN;
terminal Integer NUMBER;

non terminal expr_list, expr_part;
non terminal Integer expr;

precedence left PLUS;
precedence left TIMES;

expr_list ::= expr_list expr_part | expr_part;
expr_part ::= exprie {: System.out.println(" = "+e+";"); :} SEMI;
expr ::= NUMBER:n {
    | expr:l PLUS expr:r
    {: RESULT=new Integer(l.intValue() + r.intValue()); :}
    | expr:l TIMES expr:r
    {: RESULT=new Integer(l.intValue() * r.intValue()); :}
    | LPAREN expr:e RPAREN
    {: RESULT=e; :}
};
```
Syntactical errors are handled through the use of error productions and a special CUP error symbol which allows the parser to recover from syntax errors. This implementation is described in detail in the System Design section of this report. The culmination of the parsing phase is construction of an abstract syntax tree (AST) representing the entire Plain program. This AST is passed to future interpreter phases for use in semantic analysis and synthesis into Java source code.

Abstract Syntax Generation

The abstract syntax representation is implemented during program parsing by building structure that represents the entire program. An abstract class is defined for each general construct (i.e., statements and expressions) and a subclass is defined for the specific program construct (i.e., `new` Expression or `if` Statement). The purpose of the abstract syntax is twofold:

1) It enables a straightforward synthesis into Java source code.

2) It provides a more suitable tool than concrete syntax for future phases and development of a full-blown compiler implementation.

The Plain abstract syntax classes are as follows (a dash (“-“) indicates subclass):

```
abstract class Ast{}
abstract class Decl extends Ast{}
-ClassDecl(String name, String parentclass, DeclList d)
-MethodDecl(String name, Typ typ, FrmlArgList fal, StmtList sl)
-VarDecl(String name, Typ typ, Exp initial_value)
abstract class Exp extends Ast{}
-IntExp(String value)
```
-BoolExp(String value)
-StringExp(String value)
-NullExp()
-DerefExp(Var var)
-BopExp(Exp left, int operator, Exp right)
-MethodCallExp(Exp exp, String name, ExpList params)
-NewExp(Typ typ, ExpList params)
abstract class Stmt extends Ast{}
-AssignStmt(Var var, Exp exp)
-IfStmt(Exp test, StmtList then_clause, StmtList else_clause)
-WhileStmt(Exp test, StmtList body)
-ForStmt(Stmt initial, Exp test, Stmt increment, StmtList body)
-ReturnStmt(Exp value)
-MethodCallStmt(Exp exp, String name, ExpList params)
-DeclStmt(Decl decl)
abstract class Typ extends Ast{}
-NameTyp(String name)
-ArrayTyp(Typ elem_type)
abstract class Var extends Ast{}
-SimpleVar(int lb, int cb, String n)
-InstVar(Exp exp, String name)
-SubscriptVar(Var var, Exp index)
DeclList(Decl d, DecList dl)
ExpList(Exp e, ExpList el)
FrmlArgList(Decl d, FrmlArgList fal)
StmtList(Stmt s, StmtList sl)

The following example illustrates conversion of the simple Vehicle class into abstract syntax.

**Plain Source Code**

```java
objectClass Vehicle{
    object make := new String("");
    object model := new String("");
}
```

**Abstract Syntax Representation**

```java
AST = new DeclList(new ClassDecl("Vehicle", null,
    new DeclList(new VarDecl("make", new NameTyp("object"),
        new NewExp(new NameTyp("String"),
            null)),
    new DeclList(new VarDecl("model", new NameTyp("object"),
        new NewExp(new NameTyp("String"),
            null)),
    null),
null),
null),
null),
null)
```

**Semantic Analysis**

The first phase of semantic analysis involved adding all Plain built-in classes (and their respective methods and variables) to the AST structure. This preliminary step is necessary to ensure that any reference to these classes is properly handled during semantic analysis and type checking.
The second phase of semantic analysis consisted of the construction of symbols tables to hold program classes, methods, variables and types. Four symbol tables were required to implement this phase as depicted in Figure 2.

As depicted in the figure, each class entered into Classes table will have a corresponding entry in the type table to ensure that objects of that class can be instantiated. Each Class entry will also have a corresponding Class Variables and Methods table to hold all Class variables and methods. The Local Variables table belongs to each method—in turn—during type checking of that method’s body since local variables are limited to that scope.

![Figure 2. Symbol Table Implementation](image-url)
During construction of the tables, the following semantic checks are performed:

1) Each identifier is declared only once in the same scope with the exception of overloaded methods.
2) Overloaded methods have different signatures.
3) Overriding methods have like signatures.
4) Referenced types have been declared.

The final phase of semantic analysis consists of type checking the elements of the AST to ensure the following:

1) Each identifier referenced is represented in the symbol tree
2) Methods and variables are used in the proper scope
3) Proper types are used to compose all declarations, expressions and statements.

Synthesis into Java Source Code

The abstract syntax is converted directly to Java Source Code by stepping through the AST and converting each structure. The synthesis process is rather straightforward. Each AST construct is converted to a String representation and a corresponding Java statement is written to a Java source code file.

Execution of Java Source Code

Execution of produced Java source code is accomplished through creation of separate processes within the Java Runtime Environment to execute the Java compiler command (javac) and run the newly created class files which result.

The Plain Graphical User Interface

The Plain GUI is implemented in a Windows environment and includes a text editor for creating, editing, opening and saving Plain source-code files. Additional controls are
included for checking source-code syntax and running the source code file present in the
text-editor window. Table 5 summarizes the features of the Plain GUI. Figure 3 is a
screen capture of the Plain GUI.

Table 5. Plain GUI Menu Options.

<table>
<thead>
<tr>
<th>Option</th>
<th>Sub-option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>File</td>
<td>Open</td>
<td>Open a file browser window for local-drive navigation and selection of file containing desired Plain source code file.</td>
</tr>
<tr>
<td></td>
<td>Save</td>
<td>Save the file currently opened in the text-editor window. Prompt for a file name if the file is being saved for the first time.</td>
</tr>
<tr>
<td></td>
<td>Save As</td>
<td>Open a Save As window for local-drive navigation and entry of the desired file name and path for saving the plain source code currently in the text-editor window.</td>
</tr>
<tr>
<td></td>
<td>Exit</td>
<td>Exit the Plain programming environment and close all active windows within it. Prompt user to save current file if contents have changed since opening.</td>
</tr>
<tr>
<td>Program</td>
<td>Check</td>
<td>Invoke the Plain Interpreter; perform lexical, syntactical and semantic analysis, providing a window that lists all syntax errors by the line number on which they were encountered. Return to the text editor window without running the program.</td>
</tr>
<tr>
<td></td>
<td>Program</td>
<td>Run Program</td>
</tr>
<tr>
<td></td>
<td>Larger Font</td>
<td>Increase font size of text in the editor window by 1 point.</td>
</tr>
<tr>
<td></td>
<td>Smaller Font</td>
<td>Decrease font size of text in the editor window by 1 point.</td>
</tr>
<tr>
<td>Help</td>
<td>About</td>
<td>Display Plain Programming Environment version information in the text editor status bar.</td>
</tr>
</tbody>
</table>
objectClass Vehicle{
    object make := new String("");  
    object model := new String("");  
    boolean CHEVY := true;  
    integer odometer;  
    method construct Vehicle(String mfr,String mdl,
        integer miles){  
        make := mfr;  
        model := mdl;  
        this.odometer := miles;  
    }  
    method mileage(integer vehicleMiles){
        odometer := vehicleMiles;  
        return;
    }
}

objectClass OldTruck inherits Vehicle{
    method construct OldTruck(String mfr, String mdl,
        integer odoReading){  
        super(mfr,mdl,odoReading);  
    }  
    method mileage(integer truckMiles){
        if(truckMiles < 100000) then
            
C:\TAMUCC\Program\next\test\3.psf opened

Figure 3. Plain GUI.
ENVIRONMENT

The Plain mini-language was developed and implemented on a Personal Computer (PC), and is designed to run on the same type of machine with the following characteristics:

1) Pentium, Celeron, Pentium II or Pentium III processor operating at 333MHz or faster.
2) A minimum of 64MB RAM.
3) Windows NT or Windows 95/98 operating system.
4) JRE 1.2.2.

The Plain Graphical User Interface (GUI) and interpreter is implemented in the Java programming language using JDK 1.2.2. The text editor is a modification to an existing Java-based editor, with functionality added for checking syntax and running Plain programs. For proper operation of built-in classes, the executable Plain.jar file must be located in a directory containing the javabook package.

Two Java-based compiler development tools were used to implement lexical and syntactical analysis. The lexical analyzer generator JLex, version 1.2, was used to develop the Plain lexical analyzer, and the parser generator Constructor of Useful Parsers (CUP), version 0.10j, was used to develop the Plain parser.

---

2 Plain.java is adapted from simpleTextEditor.java, written by Andy Hoskinson.
3 Copyright 1996 by Elliot Joel Berk.
4 CUP was originally written by Scott Hudson, in August of 1995. It was extended to support precedence by Frank Flannery in July of 1996. On-going improvements have been done by C. Scott Ananian, the CUP maintainer, from December of 1997 to the present.
RESULTS

The Plain programming environment provides an easy-to-use, object-oriented learning environment. The entire package is contained within an executable Java archive (.jar) file which will launch the Plain GUI. The user can then elect to create a new program file within the GUI or open an existing Plain source file. When the user is ready, the Plain interpreter checks the syntax and semantics of the program (Check Program mode) or runs the program (Run Program mode—which also involves syntactical and semantic analysis). A pop-up text window displays syntax and semantic errors and indicates the quantity of errors encountered. Errors are listed with the corresponding line number where the error was encountered.

The Plain programming environment lacks critical functionality required in a programming application. The following is a list of essential features unimplemented in the Plain programming environment:

1. No attempt is made to monitor changes the source code file currently opened in the text editor. Should the user choose to exit the application, no attempt is made to ensure that changes are written to the existing file or that a new file is saved prior to exiting. This deficiency will cause all changes—or in some cases, entire files—to be lost upon exiting.

2. Display of the current line number in the status bar at the bottom of the editor window is required. This editor feature is crucial for debugging programs since errors will refer to the line number on which the error is encountered.

3. A print option to print the source code listing is required. Without this feature, all debugging and review of source code must be completed while the
program is loaded in the editor window. Since the Plain programming language is intended for teaching, it would be helpful if students could produce a listing to present the instructor for help with problems.

4. As currently configured, error messages are simply displayed in a pop-up window for the programmer. This window will stay open as long as necessary, but must be closed before program control is resumed by the text editor. The ability to print these errors or switch between the text editor and error window is essential.

5. Although thorough error reporting is provided in the Plain system, no mechanism is available to capture of Java compiler or runtime errors should they occur in the Execution phase. More thorough error trapping within the interpreter or ability to capture these Java errors is necessary for system stability and reliability.

The following is a list of non-essential feature that would improve the Plain programming environment:

1. The ability to undo recent changes made to the source code. An undo capability provides a powerful tool—especially to a beginning programmer—to prevent inadvertent and intentional (but mistaken) deletions and additions of source code.

2. A cut, copy and paste capability would aid in editing source code files. These options allow large blocks of text to be moved or duplicated, easing the burden of rearranging source code.
3. The absence of a help system will make first-time use of the Plain programming environment impossible without reference material available. As a mini-language, Plain syntax will not be familiar to the user and some type of introductory training will be required before efficiency can be realized.

4. Plain error reporting is currently accomplished by two separate objects—one for syntax errors and another for semantic and type errors. A unified error reporting mechanism would simplify application source code and allow interpreter progress to be centrally reported during program analysis and execution.

In its current configuration, reliable array operations are not provided. The necessary constructs are available in the \texttt{ArrayTyp} and \texttt{SubscriptVar} AST data structures, but implementation of follow-on processing for these types has been unsuccessful. While programs containing array declarations and references will not cause errors in the Plain environment, Java compiler and/or runtime errors will occur.
SYSTEM DESIGN

The Plain programming environment, Figure 4, consists of two primary modules. The Graphical User Interface provides a text editor for entry of source code and necessary control options for checking program syntax and execution of programs. The Plain Interpreter implements lexical, syntactical and semantic analyses, as well as synthesis into Java source code and subsequent execution of generated code.

The Plain Graphical User Interface (GUI)

The Plain GUI, Figure 5, controls the overall operation of the program through the Plain class. The Plain class file, Source Code Listing 1 in Appendix D, declares several variables for centralized control of all interpreter phases. Table 6 is a complete listing of these variables and their function within the Plain programming environment. All Plain Interpreter processing occurs in the `actionPerformed` method of the Plain Class. This method consists of a group of if-else if statements to determine which menu option has been selected. Interpreter action occurs when either the Check Program or Run Program options are selected.

---

5 Plain.java is adapted from simpleTextEditor.java, written by Andy Hoskinson. Modifications were made to invoke functions for interpretation, conversion and execution of Plain source code files.
Figure 5. Plain GUI - Level 2, DFD 1.1

Table 6. Plain Class Variables.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>prog</td>
<td>DeclList</td>
<td>Assigned the value of the parser result (the abstract syntax representation of the source code).</td>
</tr>
<tr>
<td>st</td>
<td>SymTable</td>
<td>Instance of SymTable to enable semantic analysis and symbol table construction. The SymTable class is described in the Semantic Analysis section of System Design.</td>
</tr>
<tr>
<td>ic</td>
<td>InitClasses</td>
<td>Instance of InitClasses to enable addition of Plain built-in classes to the AST construct prior to semantic analysis and type checking.</td>
</tr>
<tr>
<td>symbolTable</td>
<td>Hashtable</td>
<td>Hash table used to hold all built-in and user-defined classes during semantic analysis and type checking.</td>
</tr>
<tr>
<td>TypeTable</td>
<td>Hashtable</td>
<td>Hash table used to hold all predefined and user defined types during semantic analysis and type checking.</td>
</tr>
<tr>
<td>tc</td>
<td>TypeCheck</td>
<td>Instance of TypeCheck to enable type checking of source code. The TypeCheck class is described in the Semantic Analysis section of System Design.</td>
</tr>
<tr>
<td>conv</td>
<td>Converter</td>
<td>Instance of Converter to enable conversion of Plain source code to Java source code. The Converter class is described in the Conversion section of System Design.</td>
</tr>
<tr>
<td>out</td>
<td>FileOutputStream</td>
<td>Provides output stream for writing Java source code to plnprog.java file.</td>
</tr>
<tr>
<td>Variable</td>
<td>Type</td>
<td>Function</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------</td>
<td>-------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>p</code></td>
<td><code>PrintStream</code></td>
<td>Provides print stream for sending lines of Java source code to <code>plnprog.java</code> file.</td>
</tr>
<tr>
<td><code>instream</code></td>
<td><code>FileInputStream</code></td>
<td>Provides input stream for lexical analyzer to obtain source code tokens.</td>
</tr>
<tr>
<td><code>ei</code></td>
<td><code>ErrorInfo</code></td>
<td>Instance of <code>ErrorInfo</code> to simplify error counting and occurrence during semantic analysis and type checking. Consists of an integer for counting errors and a boolean to determine if errors have occurred.</td>
</tr>
<tr>
<td><code>eframe</code></td>
<td><code>Frame</code></td>
<td>Provides frame for <code>OutputBox</code> used to display semantic and type errors.</td>
</tr>
<tr>
<td><code>errOut</code></td>
<td><code>OutputBox</code></td>
<td>Instance of <code>OutputBox</code> for displaying semantic and type errors.</td>
</tr>
<tr>
<td><code>executor</code></td>
<td><code>Executor</code></td>
<td>Instance of <code>Executor</code> to enable compilation and execution of Java source code contained in <code>plnprog.java</code> file. The <code>Executor</code> class is described in the Execution section of System Design.</td>
</tr>
<tr>
<td><code>sourceFileName</code></td>
<td><code>String</code></td>
<td>Represents Plain source code file name within <code>Yylex</code> class. Filename is required by <code>TokenValue</code> class and intended for error reporting purposes but not utilized in Plain error reporting scheme.</td>
</tr>
<tr>
<td><code>bar</code></td>
<td><code>MenuBar</code></td>
<td>Instance of <code>MenuBar</code> to enable GUI menu creation.</td>
</tr>
<tr>
<td><code>fileMenu</code></td>
<td><code>Menu</code></td>
<td>Instances of <code>Menu</code> representing top-level menu selections available in the Plain GUI.</td>
</tr>
<tr>
<td><code>editMenu</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>helpMenu</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>runMenu</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>fileOpen</code></td>
<td><code>MenuItem</code></td>
<td>Instances of <code>MenuItem</code> representing choices contained in Plain GUI top-level menus. The functions of each of these options is described in the Plain GUI section of the Narrative.</td>
</tr>
<tr>
<td><code>fileSave</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>fileSaveAs</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>fileExit</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>selectAll</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>bigger</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>smaller</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>helpAbout</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>runCheckProgram</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>runRunProgram</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>myTextArea</code></td>
<td><code>TextArea</code></td>
<td>Instance of <code>TextArea</code> which holds source code text within the Plain GUI.</td>
</tr>
<tr>
<td><code>myStatusBar</code></td>
<td><code>Label</code></td>
<td>Instance of <code>Label</code> which holds Plain GUI status bar text.</td>
</tr>
<tr>
<td><code>fileOpened</code></td>
<td><code>boolean</code></td>
<td>Boolean value to determine if a file is open prior to reflecting this information in the status bar.</td>
</tr>
</tbody>
</table>
Check Program Mode

When the Check Program option is selected, the following events occur:

1. The contents of `errOut`, `symbolTable` and `typeTable` are cleared to remove data from any previous interpreter actions in the current session.
2. The contents of the text editor window are written to a temporary file—`plaintmp.psf` and `instream` is set to this file.
3. The `Yylex` and `parser` objects are instantiated with the variables `yl` and `parser_obj`, respectively.
4. The `Yylex` instance (`yl`) created is assigned as the lexical analyzer for `parser_obj`.
5. The `parser` `OutputBox` `errorOut`, used for display of syntax errors is cleared.
6. The `parser` method `parse()` is called to initiate parsing of the source code file.
7. The results of the parse are assigned to the `DeclList` variable `prog`.
8. If any syntax errors were encountered during the parse, `errorOut` remains visible, displaying the syntax error messages. When the user closes `errorOut`, Program processing is terminated and the system returns to Text Editor mode.
9. If no syntax errors were encountered during parsing, the Plain built-in classes are added to the abstract syntax through a call to the `InitClasses` method `initBuiltInClasses`. This method takes the `DeclList` `prog` and returns a `DeclList` containing concatenation of the Plain built-in classes and `prog`. This structure is assigned to a new `DeclList` `combined`. 
10. The SymTable class is instantiated with the references to OutputBox `errOut` and `ErrorInfo ei` to enable display and tracking of errors while building the hash tables.

11. The SymTable method `initTypeTable` is called to initialize `typeTable` with Plain predefined types and “void” for insertion into method definitions without a type declared.

12. The SymTable method `tableClasses` is called to initiate population of symbolTable and completion of `typeTable`. References to these two hash tables and the `DeclList` combined are passed to `tableClasses`.

13. The TypeCheck class is instantiated with references to the hash tables, OutputBox `errOut` and `ErrorInfo ei`.

14. The TypeCheck method `checkDeclList` is called to initiate type checking. A reference to the source code `DeclList prog` is passed to this method. The null value passed is replaced by a class symbol table entry for internal processing within the `checkDeclList` method.

15. The OutputBox `errOut` is updated to display the number of errors encountered. When the user closes `errOut`, the system returns to Text Editor mode.

Run Program Mode

The Run Program Mode is exactly the same as Check Program until completion of type checking. If semantic or type checking errors are encountered, `errOut` is displayed and Text Editor Mode is entered when it is closed. If no errors are detected, the following additional processing occurs:
1. The `errOut` OutputBox is hidden and cleared to prepare for program conversion and execution.

2. The `plnprog.java` file is created and opened and a `PrintStream` is instantiated to prepare for data output.

3. The import declaration is written to the file. The `javabook` package contains all Plain built-in classes.

4. The `Converter` class is instantiated and the method `convertDeclList` is invoked with reference to `DeclList prog` to initiate conversion to Java source code. The `PrintStream` is closed upon completion.

5. The status of the `ErrorInfo` is checked to determine if errors were encountered during conversion. If errors were encountered, an `errOut` is displayed with applicable messages and system processing stops without execution.

6. The `Executor` is instantiated and the method `findMain` is invoked with reference to `prog` and `symbolTable` to initiate compilation and execution of the `plnprog.java` source code. These references are required in `Executor` in order to determine which source code class contains the `main` method for the user program. If a main method is not found, `errOut` is displayed and execution is terminated; otherwise the user program is executed. When `errOut` or the user program is closed, the system returns to Text Editor Mode.
Text Editor Mode

The Text Editor mode provides a simple text editor for composition and editing of source code files. File operations are provided through `FileDialog` objects for saving and opening source code files. Text font size can be increased and decreased through use of the Smaller and Larger options under the Edit menu. These operations simply increase the font size by one point. The status bar displays the file name of currently open files and provides confirmation of file save operations.

Figure 6 depicts the State Transition Diagram for the Plain GUI.
The Plain Interpreter

The Plain Interpreter, Figure 7, consists of five modules as follows:

The *Lexical Analyzer* breaks the Plain source code into tokens. JLex is used to build the lexical analyzer and the parser generator created by CUP invokes the lexical analyzer with requests for each token. Tokens not included in the specification file result in a syntax error being issued and prevent synthesis into Java source code in the Run Program mode.

The *Syntax Analyzer* parses the Plain source code in accordance with the context-free grammar for the Plain language. Any statements not conforming to the grammar result in a syntax error being issued and prevent synthesis into Java source code in the Run Program mode. An abstract syntax tree for the source code is produced during parsing.

The *Semantic Analyzer* is invoked by the Plain class and builds symbol tables for use in type checking the Plain source code. An error is issued if any discrepancies exist, and synthesis to Java source code is inhibited in the Run Program Mode.

The *Synthesizer* converts the Plain source code directly into Java source code.

The *Executor* invokes the Java compiler for compilation and execution of the Java source code generated by the Synthesizer.

The System Structure chart for the Plain Programming Environment is depicted in Figure 8. The Plain GUI provides overall control and invokes required program analysis and execution modules based on the selected mode.
Figure 7. The Plain Interpreter Level 2 DFD 1.2.

Figure 8. Plain Programming Environment Structure Chart.
Lexical Analysis

This phase of the project involved implementation of a lexical analyzer to break the Plain language source code into tokens. The lexical analyzer “feeds” tokens to the parser. The lexical analyzer was generated through use of JLex. JLex takes a specification file composed of regular expression rules and produces a Java source code file for a lexical analyzer. Compilation of the produced source code file produces a Yylex.class file.

Plain JLex Specification

Source Code Listing 2, in Appendix D, is a complete listing of the Plain JLex specification file, Plain2.lex. The Plain JLex specification is comprised of three sections: user code, JLex directives and regular expression rules. Key components of each section are discussed in the following paragraphs.

The user code section contains the definition and a constructor for the TokenValue class. A TokenValue object is returned by the lexical analyzer when a regular expression is matched. This enables capture of line, character, filename and value (i.e., string value of matched text) within a single object. This data is later used by the parser to build data structures and for production of useful error messages. Methods for conversion to string, boolean and integer values are also included.

The JLex directives section of the specification contains directives used by JLex in production of the lexical analyzer. The following is a brief description of each directive used in the Plain2.lex specification file:

%implements java_cup.runtime.Scanner

6 TokenValue class source code obtained from UC Berkeley JLex Tutorial for CS164, Fall 1999.
Specifies that the generated Yylex class should implement the

java_cup.runtime.Scanner interface.

%function next_token

Changes the name of the Yylex tokenizing function from yylex() to

next_token().

%type java_cup.runtime.Symbol

Specifies that Yylex should return an instance of the

java_cup.runtime.Symbol class. This directive allows the parser

generated by CUP to use Yylex.

%eofval{
    return new Symbol(sym.EOF, null);
%eofval}

Specifies the value that Yylex is to return when the end of file is reached.

{%
    public String sourceFilename;
    public int lineNumber=1;
%
}

Specifies that two instance variables—sourceFilename and

lineNumber—are to be included in the Yylex class.

%line

Enables line counting so Yylex will keep track of the current line in the

Plain source file being analyzed. Line counting is zero-based and can be

accessed via the Yylex integer variable yyline.

%char
Enables character counting so Yylex will keep track of the current character in the Plain source file being analyzed. Character counting is zero-based and can be accessed via the Yylex integer variable yychar.

%state SHORT_COMMENT
%state LONG_COMMENT

Defines two lexical states—SHORT_COMMENT and LONG_COMMENT. An initial state—YYINITIAL—is implicitly defined and is the starting state by default.

The regular expressions rules section of the specification tells Yylex what action to take when an expression is matched and has the format:

```plaintext
[<state>] <expression> { <action> }
```

where state is a valid state specified in the directives section or YYINITIAL (YYINITIAL is implicit if the state is omitted), expression is the lexical rule to be matched, and action is the Java source code to be executed when a match is found. The following excerpt from the plain2.lex specification file is provided as an example.

```plaintext
<YYINITIAL>"objectClass" { return new Symbol(sym.OBJECTCLASS, new TokenValue(yytext()), yyline, yychar, sourceFilename));}
```

If the expression “objectclass” is matched while Yylex is in the YYINITIAL state, a new java_cup.runtime.Symbol is returned, consisting of the following:

- An integer constant, sym.OBJECTCLASS (defined in the CUP generated sym.class) representing the matched expression.
- A new TokenValue object comprising the matched text (“objectClass”), the current line number, the current character number and the source file name.
Lexical Analyzer Generation and Testing

JLex produces a lexical analyzer from the plain2.lex specification file through execution of the following command:

```
java JLex.Main plain2.lex
```

This results in production of a Java source file—plain2.lex.java—which contains source code for the Yylex class (see Source Code Listing 3 in Appendix D). When compiled, the Yylex.class file is produced.

Testing was accomplished through use of a test program, lextest.java, which invokes Yylex on an input file and prints a message for each matched expression.

Source Code Listing 4, in Appendix D, is the complete lextest.java file. The sym class, normally generated by CUP, is simulated to provide values for the integer constants representing expression matches (i.e., sym.OBJECTCLASS). A series of Plain source code test files, each more complex than the previous, were developed to ensure compliance with the Plain language specification.

Problems Encountered/Debugging

Two implementation difficulties were encountered during the Lexical Analyzer phase of the project, both dealing with incorrectly formed regular expression rules. The IDENTIFIER regular expression rule had to be moved toward the end of the specification to prevent Plain keywords from being interpreted as identifiers. Since JLex uses the first rule in cases where an expression matches more than one rule, expressions like “objectClass” and “integer” were being incorrectly matched to the identifier rules because it was initially located before the others.

---

7 Portions of lextest.java adapted from on-line source code examples from Modern Compiler Implementation in Java, by Andrew W. Appel.
The second difficulty applied to multi-line comments. For the \texttt{LONG\_COMMENT} state, an action for encountering \texttt{newline} was necessary to prevent an unmatched token error. The metacharacter “.” was initially used alone, but matches everything except a \texttt{newline}.

**Syntactical Analysis**

This phase of the project involved implementation of a parser to parse the Plain source code in accordance with the Plain grammar rules. The parser was generated using the Java-based Constructor of Useful Parsers (CUP). CUP takes a specification file composed of grammar rules and produces two Java source code files: \texttt{parser.java} and \texttt{sym.java}. \texttt{parser.java} contains the actual parser code and \texttt{sym.java} contains a series of constant declarations, one for each terminal symbol.

**Plain CUP Specification**

Source Code Listing 5, in Appendix D, is a complete listing of the Plain CUP specification file, \texttt{plain5.cup}. The Plain CUP specification is comprised of four sections: user code, symbol lists, precedence declarations and the grammar. Key components of each section are discussed in the following paragraphs.

The user code section is a series of optional declarations that allow user code to be included as part of the generated parser. These declarations take the form:

\begin{verbatim}
declaration_type { : ... :};
\end{verbatim}

where \{ : ... :\} is a code string whose contents will be placed directly within the appropriate class declaration. The Plain CUP specification uses two of these optional declarations: \texttt{parser code} and \texttt{scan with}. 

58
The parser code declaration contains code that will be included as part of the parser class. For the Plain Interpreter, this section is used for the following:

1. Declaration of a variable—parseResult—which will contain the entire parse tree for the Plain source code file. This variable is of type DeclList, which is one of the Abstract Syntax Tree classes discussed later in this section.

2. Declaration and initialization of an errorCount variable.

3. Instantiation of the Yylex class object yylex.

4. Overriding of CUP’s lr_parser class error reporting mechanisms to allow meaningful error reporting and prevent parse abortion when errors with unmatched productions occur.

The scan with declaration indicates how the parser should ask for the next token from the lexer, in this case the Yylex method next_token() is used.

Following the user code section, two symbol lists are provided—terminals and non-terminals. These lists name and specify the type for each terminal and non-terminal symbol which appears in the grammar and represented at runtime with java_cup.runtime.Symbol objects. Terminals are returned by the lexical analyzer and placed on the parse stack, while non-terminals replace a series of Symbol objects on the parse stack when they match a production for that non-terminal. All terminals are defined as type TokenValue (defined in the Lexical Analysis section), while non-terminals are typed according to the abstract syntax data structure they represent.
The third section specifies the associativity and precedence of terminals, and allows elimination of shift-reduce conflicts associated with ambiguous grammars. Precedence associativity declarations take the form:

```
precedence associativity terminal[, terminal...];
```

where associativity is left, right or nonassoc. The Plain CUP associativity and precedence declarations are:

```
precedence left GT, LT, EQ;
precedence left OR;
precedence left AND;
precedence left PLUS, MINUS;
precedence left MULT, DIV, MOD;
```

indicating that all operations are left associative and that the highest precedence belongs to multiplication, division and modulus division.

The final section of the Plain CUP specification is the grammar. The following excerpt from the Plain CUP specification grammar section will be used to explain the structure of productions:

```
IfStmt ::= IF:i LPAREN Exp:e RPAREN THEN LBRACE StmtList:s1 RBRACE ELSE LBRACE StmtList:s2 RBRACE
{: RESULT = new IfStmt(i.lineBegin,i.charBegin, e,s1,s2);
 :}
| IF:i LPAREN Exp:e RPAREN THEN LBRACE StmtList:s1 RBRACE
{: RESULT = new IfStmt(i.lineBegin,i.charBegin, e,s1,null);
 :}
;
```

Each production begins with a left-hand side (LHS) non-terminal (IfStmt), followed by the symbol “::=” and a series of right-hand side (RHS) productions and associated action code. If multiple productions are required the bar(\|) is used to separate them. In the example above, terminals are upper case, while non-terminals are mixed case. Terminals and non-terminals used in the RHS productions can optionally be labeled with a name. In
the example, the terminal “IF” is given then name “i”. This practice allows reference to
the labeled terminal or non-terminal within subsequent action code. Each RHS
production is optional followed by an action code, delimited by “{: :}” symbol pairs.
In the example, the CUP predefined variable RESULT is used to hold the value of a new
IfStmt abstract syntax object. The label “i” is used to provide line and character
number information because the terminal it represents (IF) was declared as type
TokenValue. While only one Java action code statement was used in the example,
multiple lines of code can be encapsulated in the action code segment. This code is
placed in the CUP generated CUP$action class within the parser.java file. The
entire production-action sequence is then terminated by a semicolon. After a completed
parse of a source code file, the parser returns a Symbol with the value instance variable
containing the RESULT of the start production, or null, if there is no value.

Parser Generation and Testing

CUP produces a parser from the plain5.cup specification file through execution of
the following command:

    java java_cup.Main < plain5.cup

This results in production of a Java source files—sym.java and parse.java—which
are provided in Appendix D as Source Code Listings 6 and 7, respectively. The sym
class contains integer constants for each terminal. Non-terminals can also be included if
the –nonterms command line option is used.

Testing was accomplished through implementation of the Plain class, which is used to
control all operations of the interpreter when the programmer selects the Check Syntax of
Run Program option. Source Code Listing 1, in Appendix D, is the complete
Plain.java file.

Error Productions

CUP supports a special error symbol which is a non-terminal that matches an erroneous
input sequence. Without this error symbol, a syntax error causes the parser to abort,
limiting the ability to produce meaningful error messages or continue the parse to
discover multiple errors in one pass. The following excerpt from the plain5.cup
specification will be used to illustrate use to illustrate error recovery:

```
AssignStmt ::= Var:v ASSIGN:a Exp:e SEMI
    {: RESULT = new AssignStmt(a.lineBegin,a.charBegin,v,e); :}
| Var ASSIGN:a error SEMI
    {: parser.report_error("Syntax Error:line " + a.lineBegin + ":illegal r-value in assignment.",null); :}
```

In the example, if an error is encountered in the Exp portion of an Assignment Statement,
the second production will be matched and symbols will be discarded until the next
semicolon is encountered. For error recovery to be successful, the parser must be able to
parse a sufficient number of tokens past the error symbol. If error recovery fails, the
parser calls the unrecovered_syntax_error method which is overridden in the
specification file to provide normal program termination while producing a general error
message to assist in determining the error. If recovery is successful, an error message is
produced by overriding the syntax_error method and parsing is continued. Error
messages are displayed through a pop-up window (javabook.OutputBox) defined
within the parser specification file.

Abstract Syntax Implementation

During source code parsing, an Abstract Syntax Tree (AST) is created through the action
code associated with each production. The abstract syntax is composed of numerous
classes to represent each distinct source code component of the Plain language. Table 7 summarizes the AST classes and their Plain language construct counterparts. Source Code Listing 8, in Appendix D, is a combined listing of all AST source files.

Table 7. Plain AST Classes.

<table>
<thead>
<tr>
<th>AST Class</th>
<th>Plain Construct/Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ast</td>
<td>Abstract parent class for all AST classes</td>
</tr>
<tr>
<td>DeclList</td>
<td>Declaration list (class, method and variable)</td>
</tr>
<tr>
<td>ExpList</td>
<td>Expression list</td>
</tr>
<tr>
<td>FrmlArgList</td>
<td>Formal argument list for method declarations</td>
</tr>
<tr>
<td>StmtList</td>
<td>Statement list</td>
</tr>
<tr>
<td>Decl</td>
<td>Abstract declaration class</td>
</tr>
<tr>
<td>ClassDecl</td>
<td>Class declaration</td>
</tr>
<tr>
<td>MethodDecl</td>
<td>Method declaration</td>
</tr>
<tr>
<td>VarDecl</td>
<td>Variable declaration</td>
</tr>
<tr>
<td>Exp</td>
<td>Abstract expression class</td>
</tr>
<tr>
<td>IntExp</td>
<td>Integer expression</td>
</tr>
<tr>
<td>BoolExp</td>
<td>Boolean expression</td>
</tr>
<tr>
<td>StringExp</td>
<td>String expression</td>
</tr>
<tr>
<td>NullExp</td>
<td>Null expression</td>
</tr>
<tr>
<td>DerefExp</td>
<td>Deferential expression for variables encountered where expressions are expected by AST classes</td>
</tr>
<tr>
<td>BopExp</td>
<td>Binary operator expression</td>
</tr>
<tr>
<td>MethodCallExp</td>
<td>Method call expression</td>
</tr>
<tr>
<td>NewExp</td>
<td>New expression</td>
</tr>
<tr>
<td>Stmt</td>
<td>Abstract statement class</td>
</tr>
<tr>
<td>AssignStmt</td>
<td>Assignment statement</td>
</tr>
<tr>
<td>IfStmt</td>
<td>If statement</td>
</tr>
<tr>
<td>WhileStmt</td>
<td>While statement</td>
</tr>
<tr>
<td>ForStmt</td>
<td>For statement</td>
</tr>
<tr>
<td>ReturnStmt</td>
<td>Return Statement</td>
</tr>
<tr>
<td>MethodCallStmt</td>
<td>Method call statement</td>
</tr>
<tr>
<td>Typ</td>
<td>Abstract type class</td>
</tr>
<tr>
<td>NameTyp</td>
<td>Name type</td>
</tr>
<tr>
<td>ArrayTyp</td>
<td>Array type</td>
</tr>
<tr>
<td>Var</td>
<td>Abstract variable class</td>
</tr>
<tr>
<td>SimpleVar</td>
<td>Simple variable</td>
</tr>
<tr>
<td>InstVar</td>
<td>Instance variable</td>
</tr>
<tr>
<td>SubscriptVar</td>
<td>Subscript variable</td>
</tr>
</tbody>
</table>

8 Most AST construct names and method definitions adapted from examples in Modern Compiler Implementation in Java, by Andrew W. Appel and UC Berkeley on-line notes for CS164, Fall 1999.
Problems Encountered/Debugging

Numerous difficulties were encountered during parser implementation which are described in the following paragraphs. Instead of calling the `lr_parser` method `parse()`, the initial implementation utilized the `debug_parse()` method. This method generates detailed status messages during parse of a source file to assist in debugging the parser.

The most difficult implementation issue was the transition from grammar rules and productions to abstract syntax. The first version of the grammar consisted of numerous productions that were very specific to the Plain construct they represented. This implementation resulted in 42 non-terminals and 134 associated productions (excluding error productions). Attempting to define a manageable abstract syntax to represent this grammar proved to be difficult due to lack of commonality between similar structures (i.e., constructor invocation and method invocation were distinct structures). The key to reducing the complexity of the AST implementation was a reduction in the number of non-terminals. The grammar was rewritten several times before an acceptable solution was derived. The final grammar resulted in only 22 non-terminals and 75 productions prior to implementation of error productions.

Problems were encountered during implementation of the `ForStmt` construct due to inconsistencies with normal Plain language rules. The initialization of the loop control variable is an implied declaration in Java. Plain requires declaration of that variable prior to its use within the `ForStmt`. A conflict arises between loop control variable initialization and normal Plain assignment statements—they are syntactically the same.
Plain documentation was updated to reflect the requirement for declaration of the loop control variable prior to its use in the `ForStmt`.

The second difficulty with the `ForStmt` construct is with the increment portion. The increment operation is essentially an `AssignStmt` without the semicolon. Rather than create a new AST construct for an assignment `expression` (and introduce another possible conflict), the Plain language description was updated to require a semicolon after the increment operation.

**Semantic Analysis**

The majority of the semantic analysis phase involved creation of symbol tables for class, method and variable identifier reference and implementation of a type checking class. Code was also required to ensure that method overloading and overriding was in conformance with Java language specifications. Additionally, Plain predefined classes and methods were added to the Abstract Syntax to facilitate reference to them in Plain source code files. The `initClasses` class—Source Code Listing 9 in Appendix D—builds the required AST for Plain predefined classes.

**Symbol Table Implementation**

Symbol table implementation was accomplished through use of four hash tables using the Java `Hashtable` class. A `typeTable` hash table was introduced for use in the type checking phase (described later). This hash table contains strings representing each valid type in the current Plain program. The `typeTable` is initialized with Plain predefined `types` (`integer`, `boolean`, `construct`, `object`, `string`) and additional `types` (`void`, `image`) to facilitate easier type checking and to support some Plain built-in types.
A symbolTable hash table was created to hold class declarations using the Symbol Table Entry (STE) abstract classes described in the next section. Within each symbolTable entry, a ClassSTE object was used to hold class definition information and another hash table—mvTable—to hold all class variable and method declarations for that specific class. A localVars hash table to hold local variables for each method was introduced in the type checking phase and is described in that section.

The STEntry Classes

To facilitate organization of the objects contained within the hash tables, an abstract Symbol Table Entry (STEntry) class was introduced. The STEntry classes were introduced to simplify storage of different Plain declaration constructs in the symbolTable, mvTable and localVars hash tables. Table 8 summarizes the Plain STEntry classes\(^9\) and describes each. Source Code Listing 10, in Appendix D, is a combined listing of all STEntry source files.

<table>
<thead>
<tr>
<th>STEntry Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STEntry</td>
<td>Abstract parent class for all STEntry classes.</td>
</tr>
<tr>
<td>ClassSTE</td>
<td>Class symbol table entry.</td>
</tr>
<tr>
<td>MethodSTE</td>
<td>Method symbol table entry. Structure facilitates implementation of a list of methods since it contains a MethodSTE variable.</td>
</tr>
<tr>
<td>InstVarSTE</td>
<td>Instance variable symbol table entry.</td>
</tr>
<tr>
<td>LocalVarSTE</td>
<td>Local variable symbol table entry.</td>
</tr>
</tbody>
</table>

\(^9\) The STEntry classes were adapted and modified from examples provided in UC Berkeley on-line notes for CS164, Fall 1999.
The SymTable Class

The SymTable class is used to build the typeTable, symbolTable and embedded mvTable(s) by stepping through the AST created in the syntactical analysis phase of the project. Only class, method and class variable declarations are processed during this pass through recursive calls to the tableClasses, tableMethods and tableVars methods, respectively. Empty typeTable and symbolTable objects are passed to the SymTable class from the Plain main method. Source Code Listing 11 in Appendix D is the SymTable class.

The tableClasses method performs the following:

1. Ensures that the current class name is not previously declared.
2. Ensures that any superclass referenced exists.
3. Adds the current class name to the typeTable.
4. Adds the current class to the symbolTable through a ClassSTE construct.
5. Calls tableMethods and tableVars to process any method or class variable declarations.
6. Calls itself to process remaining class declarations.

An error message is produced if the declaration fails to meet the criteria of (1) or (2) above.

The tableMethods method performs the following:

1. Ensures that the method return type exists in the typeTable.
2. Ensures that (only) constructor methods share the class name. Adds the method to the ClassSTE mvTable through a MethodSTE construct. If the
method name already exists, adds the current method to the end of the

MethodSTE list to facilitate overriding.

3. Ensures that overloaded methods have different signatures.

4. If a superclass exists for the current class, checks for overridden methods and
   ensures that signatures are the same.

5. Verifies that the types of the formal parameters exist in the typeTable.

6. Calls itself to process any remaining method declarations.

Error messages are produced if the declaration fails to meet the criteria in (1), (2), (4), (5)
or (6) above.

The tableVars method performs the following:

1. Ensures that the variable type exists in the typeTable.

2. Ensures that the variable name is not the same as the class or existing class
   variables and methods.

3. Adds the variable to the ClassSTE mvTable.

4. Calls itself to process any remaining variable declarations.

Error messages are produced if the declaration fails to meet the criteria in (1) or (2)
above.

Type Checking

Type checking is accomplished with a second pass through the AST by the TypeCheck
class. The TypeCheck class is composed of numerous methods—one for each AST
structure—which facilitate or perform type checking on all Plain program elements.

Table 9 summarizes the TypeCheck class methods. Source Code Listing 12 in
Appendix D is the TypeCheck class.
Table 9. TypeCheck Methods\textsuperscript{10}.

<table>
<thead>
<tr>
<th>TypeCheck Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>checkDeclList</td>
<td>Calls checkDecl for each declaration in a DeclList.</td>
</tr>
<tr>
<td>checkDecl</td>
<td>Overloaded method with derivatives to determine and process each Plain declaration construct.</td>
</tr>
<tr>
<td>checkExpList</td>
<td>Calls checkExp for each expression in an ExpList.</td>
</tr>
<tr>
<td>checkExp</td>
<td>Overloaded method with derivatives to determine and process each Plain expression construct.</td>
</tr>
<tr>
<td>checkStmtList</td>
<td>Calls checkStmt for each statement in a StmtList</td>
</tr>
<tr>
<td>checkStmt</td>
<td>Overloaded method with derivatives to determine and process each Plain statement construct. In the case of a DeclStmt, the variable is added to the localVars hash table.</td>
</tr>
<tr>
<td>checkVar</td>
<td>Overloaded method with derivatives to determine and process each Plain variable construct.</td>
</tr>
<tr>
<td>checkTyp</td>
<td>Overloaded method with derivatives to determine and process each Plain type construct.</td>
</tr>
<tr>
<td>tableArgs</td>
<td>Adds each method formal argument to the localVars hash table.</td>
</tr>
</tbody>
</table>

The localVars hash table used to hold local variables within each method is created during TypeCheck object instantiation. Variables are added to the table whenever a DeclStmt is encountered in the method body (StmtList). The contents of the localVars table are cleared prior to processing each new method declaration since they are only visible within the current method scope.

Problems Encountered/Debugging

The first implementation of the semantic analysis phase involved building a symbol tree. The structure was a list of trees—one for each class—with the left branch representing class variables and the right representing methods. Within each method, the left branch represented local variables while the right represented addition methods within the same class. While implementation of this structure was successful, navigation and extraction

\textsuperscript{10} Structure of TypeCheck class derived from AST printing method provided on-line for Modern Compiler Implementation in Java, by Andrew W. Appel
of required data elements during type checking proved to be difficult and inefficient. The hash table implementation allows access to any required class from any point within the program.

Even with the hash table implementation, additional code and restructuring of the otherwise straight-forward type checking code was necessary to enable inheritance. Variables and methods not found within the current program scope required a search of outer scopes before errors could be confirmed. The use of `super` or `this` had to be isolated to determine the correct class within which to implement type checking, adding to the complexity of the `TypeCheck` class.

**Conversion**

Conversion to Java source code is accomplished by a third pass through the AST. During this pass, Plain language constructs are converted to their Java counterparts in a relatively straightforward manner by the `Converter` class. The results of conversions are written to an output file—`plnprog.java`—for future processing by the Executor phase.

Table 10 summarizes the `Converter` class methods. Source Code Listing 13 in Appendix D is the `Converter` class. The sole error condition reported by the Converter class is object type variables that have not been initialized with an actual class type assignment. Failure to trap this condition would result in Java compile errors.

**Table 10. Converter Methods.**

<table>
<thead>
<tr>
<th>TypeCheck Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>convertDeclList</td>
<td>Calls <code>convertDecl</code> for each declaration in a <code>DeclList</code>.</td>
</tr>
<tr>
<td>convertDecl</td>
<td>Overloaded method with derivatives to determine type and write each declaration to the <code>plnprog.java</code> file.</td>
</tr>
<tr>
<td>convertExp</td>
<td>Overloaded method with derivatives to determine type and convert each Plain expression construct to a String.</td>
</tr>
<tr>
<td>TypeCheck Method</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>convertStmtList</td>
<td>Calls convertStmt for each statement in a StmtList.</td>
</tr>
<tr>
<td>convertStmt</td>
<td>Overloaded method with derivatives to determine type and output each Plain statement construct to plnprog.java.</td>
</tr>
<tr>
<td>convertVar</td>
<td>Overloaded method with derivatives to determine type and convert each Plain variable construct to a String.</td>
</tr>
<tr>
<td>convertTyp</td>
<td>Overloaded method with derivatives to determine type and convert each Plain type construct to a String.</td>
</tr>
<tr>
<td>convertArgs</td>
<td>Converts a FrmlArgList to a String.</td>
</tr>
</tbody>
</table>

**Execution**

Execution of the generated Java source code is accomplished in the **Executor class**,

Source Code Listing 14 in Appendix D. The **Executor class** contains only one method—**findMain**—which performs three functions:

1. Invokes the Java compiler (**javac**) on the **plnprog.java** source file created in the Conversion phase.

2. Determines the first user-defined class that contains a **main** method. If no class containing a **main** method is found, an error message is displayed.

3. Invokes the Java bytecode interpreter (**java**) on the class containing the **main** method.

The **javac** and **java** processes are run from within the **Executor class** through an instance of the **Exec class**.\(^\text{11}\) The **Exec.execWait** method call invokes a new process (e.g., “**javac plnprog.java**”) and waits for completion of that process before calling program continues. The **Exec class** is documented in Source Code Listing 15, Appendix D.

---

\(^\text{11}\) The **Exec class** was written by Marty Hall and appears in *Core Web Programming* from Prentice Hall Publishers, 1997.
SUMMARY AND FUTURE WORK

The Plain language provides an introduction to OOP concepts and programming by including OOP terminology as part of the language. Minimal data types and program structures enable rapid understanding and simple program development. The Plain GUI is implemented in a Windows environment and features text editing capabilities in addition to controls for checking source-code syntax and running Plain programs. The implementation of the interpreter is in Java, and introduces the processes involved in OO compiler construction. The Java-based JLex and Constructor of Useful Parsers (CUP) were used to generate the Plain lexical-analyzer and parser, respectively. Construction of an abstract syntax during parsing of the Plain source code produced a data structure well suited for semantic analysis and type checking be future interpreter modules. Informative error messages are produced and displayed to assist in correction of source code errors. The interpreter accepts Plain source code, analyzes it, converts it into Java source code and executes it with the Plain programming environment. Several predefined classes are supplied to assist in the rapid development of functional programs.

The Plain Programming Environment is acceptable for its intended purpose, however, the following improvements are necessary to meet minimal user requirements:

- Monitoring of source code file changes to ensure that the programmer is afforded an opportunity to save existing work in the event of accidental program termination.
• Addition of a print option to allow generation of hard copies of source code for off-line analysis and debugging.
• Line number tracking and display to aid in debugging of source code syntactical and semantic errors.
• A unified error reporting mechanism to allow centralized status reporting and simplify the application through modularity.
• Proper processing of Array types.
• More thorough error trapping or ability to catch Java compile-time and run-time errors.

The Plain programming environment and associated source code modules provide a framework for future object-oriented interpreter and compiler implementations. The modularity of the application allows introduction additional languages though manipulation of the JLex and CUP specification files. Additionally, the Plain interpreter could easily be extended to a full-blown compiler implementation through replacement of the conversion and execution phases. The Jasmin\textsuperscript{12} assembler for the Java Virtual Machine would be one possible choice for this extension. Jasmin takes ASCII descriptions for Java classes, written in assembler-like syntax using the Java Virtual Machine instructions set, and converts them into binary Java class files.

\textsuperscript{12} Jasmin was created by Jonathan Meyer as a companion to the book \textit{Java Virtual Machine}, written by Jon Meyer and Troy Downing and published by O'Reilly Associates.
APPENDIX A – PLAIN RESERVED WORDS

AND

boolean

construct

Clock

DrawingBoard

else

for

if

inherits

InputBox

integer

main

MainWindow

MessageBox

method

new

object

objectClass

OR

OutputBox

ResponseBox

return

SketchPad

String

super

then

while
APPENDIX B – PLAIN BNF

<program>   →  <ClassDecls>
<ClassDecls> →  <ClassDecl><ClassDecls>
    |  <ClassDecl>
ClassDecl   →  objectClass <identifier> [inherits <identifier>]
              {[[<VarDecls>][<MethodDecls>]]}
(MethodDecls) →  <MethodDecl><MethodDecls>
   |  <MethodDecl>
(MethodDecl) →  method [<Typ>] <identifier> {[[<FrmlArgList>]]}
              {StmtList}
<VarDecls>   →  <VarDecl>;<VarDecls>
   |  <VarDecl>
<VarDecl>    →  <Typ><identifier>[:= Exp]
              |  <identifier><identifier>[:= Exp]
<Typ>       →  integer | boolean | construct | object | <Typ>[]
<FrmlArgList> →  <VarDecl>, <FrmlArgList>
   |  <VarDecl>
<StmtList>   →  <Stmt><StmtList>
   |  <Stmt>
<Stmt>       →  <AssignStmt>
              |  <IfStmt>
              |  <WhileStmt>
              |  <ForStmt>
              |  <ReturnStmt>
              |  <MethodCallStmt>
              |  <DeclStmt>
<AssignStmt> →  <Var> := <Exp>;
<Var>        →  <identifier>
              |  <Exp>.<identifier>
              |  <Var>[<Exp>]
<IfStmt>     →  if (<Exp>) then {<StmtList>}{else
              {<StmtList>}}
<WhileStmt>  →  while (<Exp>) {<StmtList>}
<ForStmt>    →  for (<stmt>;<Exp>;<stmt>) {<StmtList>}
<ReturnStmt> →  return [<Exp>];
<MethodCallStmt> →  [<Exp>].<identifier> {[[<ExpList>]]};
<DeclStmt>   →  <VarDecl>;
<ExpList>    →  <Exp>,<ExpList>
              |  <Exp>
<Exp>        →  integer-literal
              |  boolean-literal
              |  string-literal

13 Terminals are indicated with boldface type.
(null-literal)   | (null-literal)
(Exp)  →  (Var)
         |  new (Typ) ([(ExpList)])
         |  new (identifier) ([(ExpList)])
         |  (Exp).(identifier)([(ExpList)])
         |  (Exp) > (Exp)
         |  (Exp) < (Exp)
         |  (Exp) = (Exp)
         |  (Exp) OR (Exp)
         |  (Exp) AND (Exp)
         |  (Exp) + (Exp)
         |  (Exp) - (Exp)
         |  (Exp) * (Exp)
         |  (Exp) / (Exp)
         |  (Exp) % (Exp)
         |  (Exp)
(boolean-literal) →  true | false
(string-literal) →  "$"string"
(string) →  (letter | digit)
         |  (letter | digit)string
(null-literal) →  null
(identifier) →  (letter)
         |  (identifier)letter
         |  (identifier)_letter
         |  (identifier)_digit
(integer-literal) →  (digit) | (integer-literal)digit
(letter) →  A | B | C | D | E | F | G | H | I | J | K | L
         |  M | N | O | P | Q | R | S | T | U | V | W | X
         |  Y | Z | a | b | c | d | e | f | g | h | i | j
         |  k | l | m | n | o | p | q | r | s | t | u | v
         |  w | x | y | z
(digit) →  1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0
APPENDIX C – PLAIN SYNTAX CHARTS

program

ClassDecl

Class Decl

objectClass

identifier

inherits

identifier

{VarDecl ; MethodDecl }
ReturnStmt

DeclStmt

Exp

boolean-literal

true
false
APPENDIX D – SOURCE CODE

The CD-ROM attached to the following page contains all source code for the Plain project.