PCL

Portable Charting Language

May 9th 2006

Chee Seng Choong Thomas Chou

Chapter 1: Introduction

1.1 Background

Charts and graphs have been indispensable tools since the beginning of time. They are used everywhere and by everyone. In the financial field, stock brokers analyze large amounts of financial data every day to determine what to buy and what to sell. In the medical field, clinicians look at patients' charts to determine the proper treatment. In the scientific and engineering fields, charts can be used to analyze massive amounts of scientific or performance data gathered over time revealing to their users trends and anomalies that they couldn't have discovered through other means.

Today, there are a plethora of charting tools out there each tailored for a specific purpose. But ultimately, all charting users are looking for the same thing and that is how to take a collection of data and produce a meaningful chart quickly and easily. Today's charting applications can't give you that. Spreadsheet applications such as Microsoft Excel are great for data entry but can be cumbersome when manipulating large amounts of data. Furthermore, it can't do batch processing - quickly generating charts from a series of input files. 3rd party charting libraries are highly configurable but require users to have a programming background to use them.

1.2 Goals

The purpose of PCL is to address the deficiencies that exist in today's charting tools described above. The language is designed to give users the ability to build charts customized to meet their individual needs without having them to understand the technical details behind writing a charting program.

1.2.1 Architecture Neutral and Portable:

PCL is an interpreted language and will execute on top of the Java Virtual Machine. Like Java, PCL is designed to run on any platform allowing users to produce charts in almost any environment.

1.2.2 Ease of use:

PCL provides the facilities to analyze, manipulate, and extrapolate the data by just writing a few lines of code. It hides the mundane details so that the user can concentrate on manipulating and analyzing the data itself. For example, the process of opening a file, creating an input stream, tokenizing, and parsing is simplified to just specifying the filename and the delimiter(s) used.

Function loops allow the programmer to take control of repetitive tasks such as deriving a new data series from existing data as well as enable batch processing of multiple charts using the same actions. Highlighting data points that exceed specific thresholds on the chart can be expressed in a little conditional statement.

1.3 Main Language Features

The following are a subset of the main features of the PCL language:

1.3.1 Data Types

The basic data types such as integer and double are supported for data values.
Strings, which are mainly used as labels for data series, columns, and the chart's title, are also supported.

1.3.2 Basic Data Series Operations

The basic common operators such as +, -, *, /, and % are supported. Relational and boolean expressions are also allowed.

1.3.3 Program flow control

Basic flow control statements such as if-else, for, continue, return, and break are available. These make it possible to analyze and selectively manipulate data. Users will also be allowed to create their own functions.

1.3.4 Internal functions

Functions such as load, save, print, plot and add are available to manipulate .csv data and eventually plotting the results to a graph. Special validator and generator functions are used to validate and modify existing data, as well as creating new data series.

Chapter 2: Tutorial

2.1 Getting Started

PCL can be launched as an interpreter or as a compiler if a source file is given. For the purpose of this tutorial, the examples will be done in interpreter mode. In most cases, using a source file is more efficient.

To launch as an interpreter, simply type:

```
%os prompt%> java PCLMain
Pcl>
```

To launch with an input source file, just add the file's name as an argument:

```
%os prompt%> java PCLMain tutorial.txt
```

2.2 Variables

In PCL, variables are declared by the `let` keyword. The following example assigns a number to variable `i`:

```
Pcl> let i = 1;
i = 1
Pcl>
```

The interpreter will return the value of `i` again followed by the prompt. After a variable is declared, its values can be changed by simply doing the following:

```
Pcl> i = 2;
i = 2
Pcl>
```

In addition to numbers, variables can be assigned strings as well as the value of other variables. For example:

```
Pcl> let j = "this is j";
j = this is j
Pcl> i = j;
i = this is j;
```

2.3 Expressions

Basic arithmetic expressions as well as relational expressions are supported. They follow the same syntax as Java:

```
Pcl> i = 1 + 2;
i = 3

Pcl> j = 3;
j = 3

Pcl> i = i * j;
i = 9

Pcl> let t = j > i;
t = false;
```

2.4 Flow Control

Program flow control in PCL is done via `if-else`, `for`, `continue`, and `break` statements. Here are examples of an if-else and a for-loop statement:

```
Pcl> if (j > i)
print "won't see this";
else
print "not true";
not true
Pcl>

Pcl> for (i=0; i<3; i=i+1)
print i;
i = 0
i = 1
i = 2
i = 3
Pcl>
```

2.5 Loading and plotting from a CSV file

2.5.1 Loading a .csv file

To begin manipulating charts, one has to first load a .csv file. This can be done using the special chart function: `chart.load("contivity.csv"),` where contivity.csv is the filename.

2.5.2 Adding a new series to be plotted

To plot something from the data read from the .csv file, a new data series has to be defined using the function `chart.addseries()`. More than one series can be plotted to the same graph. The arguments required are:

a) The series' name
b) The column name from the .csv file for the x-axis
c) The column name from the .csv file for the y-axis
d) The choice of line, bar, or scatter plot

For example, here's the statement to create a new series named "Sessions", with columns "timestamp" and "ipsecsessions" used for the x and y axis respectively, and to plot it as a line graph:

```
chart.addseries("Sessions", "timestamp", "ipsecsessions",
"Line");
```

2.5.3 Plotting the series

To plot the series, simply use the statement:

```
chart.plot();
```

2.6 Validating data

To validate the data from the CSV, one has to first define a validating function. This function can also be used to modify the data loaded before plotting them. The statements within the validating function are executed on every row of the specified column. The row in each iteration is referenced by the special identifier `value`. For example, the following function named `isPercent` that checks if all the values are within 0 and 100:

```
[isPercent]
{
      if (value >= 0 && value <= 100)
            return true;
      else
            return false;
}
```

This validating function can then be called by passing in the defined validator's name and the column name to be validated:

```
chart.validate(isPercent, "cpuusage");
```

Here's another example of a validating function changing the data in a column. Note that the function must always provide a return statement. If none is specified, the return value of `false` is assumed:

```
[salesTax]
{
      value = value * 1.05;
      return true;
}

chart.validate(salesTax, "price");
```

2.7 Generating data

The other major function in PCL is generating new data series derived from the input data. The generator function definition is similar to the validator's except that it accepts input arguments. Inside the generator, the statement `newseries.add` is used to fill in data for the new data series. The following is an example of a generator named `ratio` that will be used to plot a new series that shows the ratio of CPU usage to IPsec sessions:

```
[Ratio]
in(time, cpu, sessions)
{
	for(let i = 0; i < time.length(); i = i + 1)
	{
		newseries.add(time[i], (sessions[i]/cpu[i])*10);
	}
	return true;
}
```

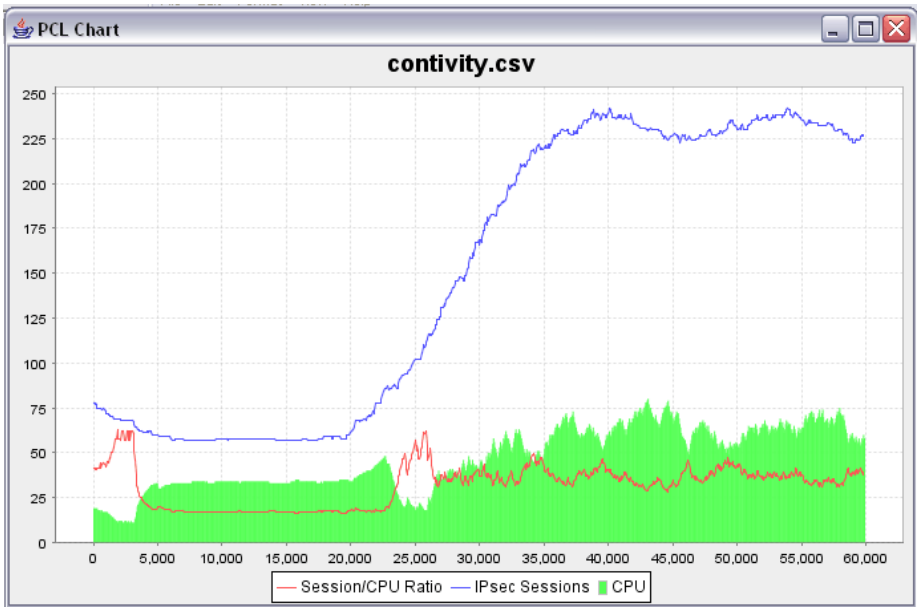
This generator function is invoked using `chart.generate` with the following arguments:
a) Generator's name.
b) The column name for the new series generated.
c) One or more list of column names to be passed in to the generator function.

Example:

```
chart.generate(Ratio, "ratio", "Line", chart.dataseries["CPU"].x,
chart.dataseries["CPU"].y, chart.dataseries["IPsec Sessions"].y);
```

2.7.1 Putting them all together

Here's how a plot would look like after putting together some of the snippets of examples above:



Chapter 3: Language Reference Manual

3.1 Lexical Conventions

The language is classified into 6 kinds of tokens: identifiers, keywords, constants, strings, expression operators and other operators. Whitespace (blanks, tabs, and newlines) are ignored and are expected to serve as token separators.

3.1.1 Comments

Comments begin with "//" and ends with a carriage return or new line. Comments can begin anywhere in a line.

3.1.2 Identifiers

An identifier consists of a sequence of letters, digits, and underscores "_". The first character of an identifier should be a letter. Identifiers are case-sensitive.

3.1.3 Keywords

The following identifiers are reserved as keywords, and should not be used otherwise:

```
if          else        for         value       return
break       continue    load        save        let
print       true        false       in          newseries
```

3.1.4 Constants

There are two types of constants – numbers and strings. Their format is defined as follows:

3.1.4.1 Numbers

A number consists of one or more digits with an optional decimal point ".". A number with a decimal must have the following format: An integer constant followed by a decimal point and an "e" or "E" followed by a signed integer exponent. A positive exponent may be preceeded by a plus sign "+". A negative exponent must have a minus sign "-".

3.1.4.2 Strings

A string can be one or more characters enclosed in double quotes " " ". Any double-quotes within a string must be preceded by a backslash "\" character.

3.1.6 Other tokens

The following operators or symbolic characters are used:

```
{     }     [     ]     (     )     ,     ;
+     -     *     /     %     =     >     <
>=    <=    ==    !=    !     &&    ||
```

3.2 Types

The types supported in PCL are Java primitives. Type checking will be done only at run time.

3.3 Expressions

3.3.1 Primary Expressions

Primary expressions involve identifiers, constants, function calls, access to data series and charts, including expressions surrounded by “(“ and “)”.

3.3.2 Identifier

An identifier is a left-value expression. It will be evaluated to a value that is bounded to this identifier.

3.3.3 Constant

A constant is a right-value expression. It will be evaluated to the constant itself.

3.3.4 (expression)

An expression enclosed in parentheses is a primary expression whose value are identical to the expression itself. The parentheses denotes precedence in the evaluation of the expression that is higher than the other expressions in the statement.

3.3.5 Function Call

A function call consists of a function identifier followed by parentheses containing one or a comma-separated list of expressions which are considered the arguments to the function.

3.3.6 Unary operators

Expressions with unary operators group left to right.

3.3.6.1 -expression

The result is the negative of the expression. This is applicable to number constants only.

3.3.7 Multiplicative operators

The multiplicative operators *, /, and % group left to right.

3.3.7.1 expression * expression

The binary * operator indicates multiplication.

3.3.7.2 expression / expression

The binary / operator indicates division.

3.3.7.3 expression % expression

The binary % operator yields the remainder from the division of the first expression by the second. Both operands must be integers.

3.3.8 Additive operators

The additive operators + and – group left to right. Precedence is deferred to multiplicative operators.

3.3.8.1 expression + expression

The result is the sum of the expressions. If one of the operands have an exponent, the result will be represented in that format.

3.3.8.2 expression – expression

The result is the difference of the operands.

3.3.9 Relational expression

Relational operators can only have two operands and are of the following types:

expression > expression (more than)
expression < expression (less than)
expression >= expression (more than or equals)
expression <= expression (less than or equals)

The operators yield the boolean value of true if the specified relation is true; false otherwise. Only numbers may be compared.

3.3.10 Equality operators

The == (equal to) and the != (not equal to) operators have lower precedence than relational operators. Otherwise they are evaluated the same way as relational expressions.

3.3.11 expression || expression

The || operator returns the boolean logical true if either of its operands is true. It groups left to right. Operands must contain only boolean values.

3.3.12 expression && expression

The && operator returns true if both of its operands are true. Like ||, it groups left to right and only accepts operands that contain boolean values.

3.3.13 Assignments

In the form of expression = expression, it assigns the value of the right operand to the left operand. The left operand has to be an identifier.

3.3.14 Declarators

A declaration begins with the keyword `let` followed by a single identifier.

3.3.15 Statements

Statements are executed in sequence, unless flow control statements indicate otherwise. Syntax notation: syntactic categories are indicated in *italics*, and literal words and characters in `gothic`.

3.3.15.1 Expression statement

Most statements are expression statements of the form
`expression;`

3.3.15.2 Compound statement

Statements can be grouped together by enclosing them with curly bracers “{“ and “}”.

3.3.15.3 Conditional statement

A conditional statement can be of the two following forms:

`if (` *expression* `)` *statement*
`if (` *expression* `)` *statement* `else` *statement*

In both cases, the expression is evaluated first. If the value is a boolean true, the first statement is executed. Otherwise, the second statement is executed. The “else” ambiguity is resolved by connecting an `else` with the last encountered elseless `if`.

3.3.15.4 For statement

The for statement has the form

`for (` *assignment / declarator* `;` *expression2* `;` *expression3*) statement

The first assignment or declarator specificies the initialization of the loop. The second expression specifies the condition that would terminate the loop. The third expression specifies the incrementation performed at the end of each iteration of the loop. All of the expressions are optional.

3.3.15.5 Break statement

The statement

```
break ;
```

causes an abort of the smallest enclosing `for` loop statement. Control passes to the statement that is immediately after the loop. This statement is not supported in the Validator and Generator functions.

3.3.15.6 Continue statement

The statement

```
continue ;
```

causes control to pass to the end of the current iteration of the smallest enclosing `for` loop. This statement is not supported in the Validator and Generator function.

3.3.15.7 Return statement

A return statement,

`return` expression `;`

is used inside the validator or generator function body. It must return either a boolean value of true or false, or an identifier containing such a value.

3.4 Predefined Objects and Functions

Like lanugages such as Javascript and VBscript, PCL comes with a set of pre-defined objects and functions used for configuring the chart before it is plotted. These pre-defined objects and functions are listed below and explained in detail in the following sections:

Object	**Properties**	**Functions**
`Chart`	`title`	`plot`
	`datacolumn`	`load`
	`dataseries`	`addseries`
`Dataseries`	`id`	`add`
	`x`	
	`y`	
	`type`	
`DataColumn`	`id`	
	`data`	

Global functions:

Function name	**Parameters**	**Description**
`print`	*string*	Prints the string to console

3.4.1 Collections

A Collection represents a container that holds a group of objects. Some of the properties defined above, such as dataseries and datacolumn, are Collections.

Square brackets "[" and "]" will be used to access the individual objects within a Collection. Some Collections can only be accessed by using a string.

Example:

```
dataseries["series name"]
```

Others such as arrays can be accessed by an integer index.

Example:

```
dataseries.x[0]
```

All collections have a `length()` that that returns the size of the collection

3.4.2 Chart

There is one and only one Chart object. It is a global variable and is named `chart.` Data for the chart object is loaded using the `load()` function which takes in the filename of the charting data file.

Example:

```
chart.load("c:\mydatafile.csv");
```

The data file must be in a csv format. Once `load()` is called, the contents of that CSV file are loaded into the Chart object. The first row of the CSV file must be the column header names. The properties of the Chart object are described below.

Property Name	**Description**
`title`	The title of the chart. Name of the csv data file loaded
`datacolumn`	A Collection DataColumn objects that represent contents of the loaded csv charting data file. Access to this collection is by name.
`dataseries`	A Collection of DataSeries objects. items are accessed by data series name.

The following functions can be applied on the Chart object.

Function Name	**Parameters**	**Description**
`addseries`	series name, x, y, type	Creates a new Dataseries object to be plotted. x and y are the names of the data columns used to be the x and y values of the dataseries. type represents the type of dataseries that will be plotted. (Scatter, Line, Bar).

`plot`		Plot all data series objects defined for this chart
load	filename	Loads data from a csv file into the chart

3.4.3 Dataseries

A Dataseries object represents a collection of data points that are to be plotted with a unique symbol and color. A user creates and modifies Dataseries objects by using the following properties and functions.

Property Name	**Description**
`id`	A unique identifier
`x`	X values collection
`y`	Y values collection
`type`	Describes the type of plot this data series will be. The possible types are {Scatter, Line, Bar}

Function Name	**Parameters**	**Description**
`add`	x, y values	Adds a data point to the data series

3.4.4 DataColumn

The csv data in the Chart object is stored as a matrix. To access the contents of the matrix, the programmer must use DataColumn object and its properties.

Each DataColumn object in the Chart's DataColumn Collection maps to a column in the matrix and the name is taken from the column names loaded from the csv file.

Property Name	**Description**
`id`	A unique identifier
`data`	An array of values loaded from the csv file

The following a sample program that shows how the objects described above are used

```
// Load the csv file and create the Chart object
chart.load("data.csv")


// Create a new data series
chart.addseries(“new series”, “x”, “y”, “scatter”);

// Plot the chart
chart.plot()
```

3.5 Validators and Generators

3.5.1 Validators

PCL has built in constructs to help users validate their data before plotting it. These constructs are called Validators and are defined in the following format:

```
[Validator-name]
{
      // Describe Validation rules here
}

chart.validate(validator-name, column-name)
```

As shown above, the declaration of a Validator must begin with the name of the validator enclosed in square brackets, "[" and "]", and the body of the Validator must be enclosed within curly braces, "{" and "}". Within the body of a validator users are allowed to use expressions and identifiers to implement their validation rules. The Validator will always return Boolean value. A return value of true will indicate that validation for current cell was successful and false otherwise. If no return statement is supplied then it is assumed that the validation failed. To validate a specific data column, the user would call the `validate()` function passing in the validator name and name of the column to be validated.

Example:

```
      chart.validate(MyValidator, "Y-Values")
```

The rule defined within the body of Validator is applied to all values within the specified data column. A special identifier, `value`, is used to represent the contents of the current cell in the column being validated. If all values in the column checked return true then the validate() function will return true; otherwise it will return false. User variables declared outside of the Validator are not accessible within body of the Validator. Variables declared within the Validator are not accessible outside the body of the Validator.

Example:

```
[MaxValueCheck]
{
      if(value > 300)
            return false
      else
            return true
}
```

Validators can also be used to correct mal-formatted or out of range values by changing the value identifier.

Example:

```
[Normalize]
{
```

```
	// Put the ceiling at 300
	if(value > 300)
		value = 300

	return true
}
```

3.5.2 Generators

A Generator allows users to create new data points from existing data points. The new data points are then used to produce a new DataSeries object that is added to the Chart. The format of a Generator is shown below.

```
[Generator-Name]
in(parameter-list)
{
	// Body of the Generator
}
```

As shown above, the declaration of a Generator begins with the Generator name enclosed in square brackets. This is then followed by the parameter declaration. The parameter declaration must begin with the `in` keyword with each parameter declared separated by commas and within the parentheses.

The body of the Generator must be enclosed with curly braces, “{“ and “}”, and the Generator must be terminated with a return statement.

To execute a generator, the user will call the generate function passing in the Generator name, the new dataseries’s name, the plot type, and a list of parameters:

```
chart.generate(generator-name, dataseries-name, plot-type,
parameter-list)
```

Within the body of the Generator, besides using regular statements and expressions, users are allowed to use the special identifier `newseries` to produce their new DataSeries. The `newseries` identifier is a DataSeries object and if the Generator call succeeds without error and returns true, this DataSeries will then be added to the Chart's DataSeries collection. Otherwise, the DataSeries is discarded.

Example:

```
[Average]
in(high, low)
{
	// Create a new series that represents the running average
	for(let i=0; i<high.x.length(); i++)
	{
		newseries.add(high.x[i], (high.y[i] + low.y[i]) / 2);
	}

	return true;
}
```

Generators have the same scoping rules as Validators.

Chapter 4: Project Plan

4.1 Team Responsibilities

Chee-Seng Choong – Lexer, Parser, Walker design and implementation.
Thomas Chou – Interpreter, Chart control interface design and implementation.

All members of the team participated in testing and preparation of the documentation.

4.2 Programming style

Standard Java coding conventions were used. Since the IDE used are identical, the coding conventions are basically left to the default used by Eclipse.

4.3 Project Timeline

Project Milestones

2/7	-	Language White Paper
3/2	-	LRM and Grammar definition
3/10	-	Evaluated and identified suitable 3rd party charting component
4/3 week	-	Lexer and Parser implementation (Chee Seng), Chart Control (Tom)
4/10 week	-	Unit testing and debugging
4/17 week	-	Walker and Interpreter implementation
4/24 week	-	Integration
5/1 week	-	Testing and Documentation

4.4 Software Development Environment

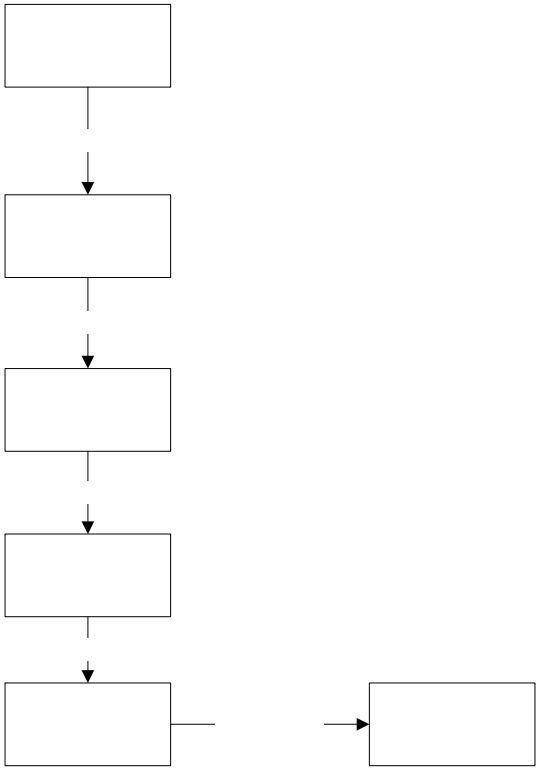
OS Platform:	Windows XP
Programming Language:	Java 1.5
IDE:	Eclipse 3.1 (http://www.eclipse.org/)
	ANTLR plugin for Eclipse (http://antlreclipse.sourceforge.net/)
Source Control:	CVS 1.11.17 on Mandrake Linux 10.1

ANTLR 2.7.6 is used for generating Parser, Lexer, and Walker.

Chapter 5: Architecture

5. 1 PCL Components

The diagram below shows the components within the PCL Interpreter.



PCLMain – The entry point for the interpreter. It accepts user console inputs or a file containing PCL expressions as an input. Data collected from the console or file are forwarded to the PCLLexer as character stream

PCLLexer – Takes the character stream and produces a stream of tokens.

PCLParser – Takes token stream and builds an AST tree

PCLWalker – Walks the AST tree produced by the PCLParser and executes functions on the PCLInterpreter.

PCLInterpreter – Acts as a proxy between the AST Walker and the Charting control. Actions defined in the AST walker are executed within this object. These actions may trigger function calls on the Charting Control.

PCLChart – The Charting control class. It handles all the logic for creating and modifying the chart.UI component.

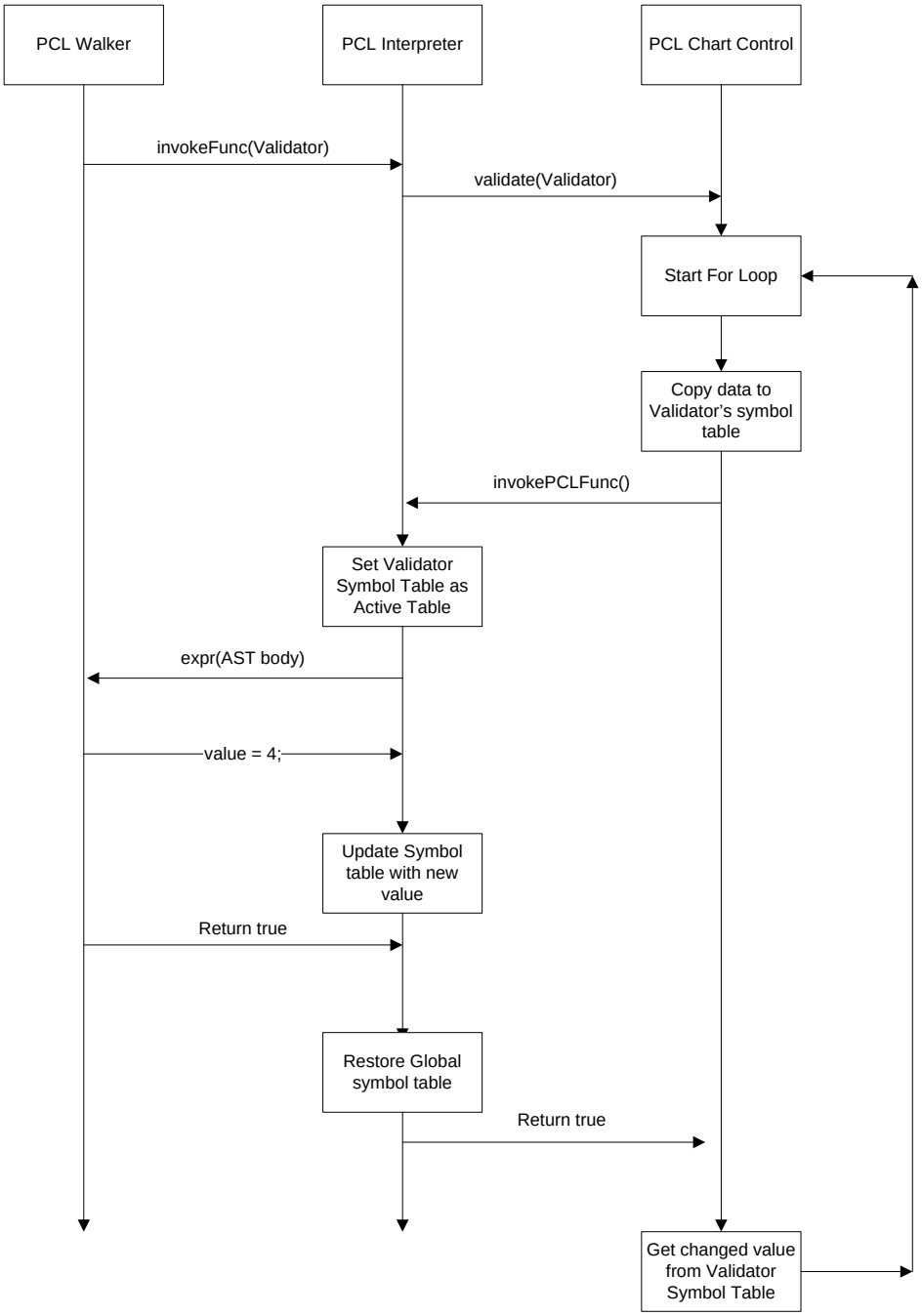
5.2 Chart Control Interface

In a PCL program, there is one and only one chart control. Users can use the chart control to add new data series objects and perform data validation. Access to chart control properties and functions is different from accessing variables and functions defined in a PCL program. Unlike variables defined in a PCL program, Chart control properties and functions are not defined in the program's symbol table.

Getting chart properties and invoking chart functions are done using Java Reflection methods. Furthermore, chart control properties are read only. Invoking the chart control's validate and generate functions is also very different from invoking a regular function. Below is a event diagram that show what happens when a Validator is called within a PCL program

```
[MyValidator]
{
	if(value > 4)
		value = 4;
	return true;
}

chart.validate(MyValidator, "ColA");
```



As you can see from the diagram, part of the function evaluation is done at the chart control level and the part of it is executed at the PCL program level. The reasons behind implementing the Charting control interface this way are

1. Prevent users from corrupting the integrity of the chart control by controlling access to key components within the chart control.
2. Hide implementation details of constructing a chart from the user.

Chapter 6: Test Plan

Test cases were developed as new features are added. These cases are then accumulated and the test results saved. These results are later file compared (using fc) against the one generated by the latest changes to the project. This is automated using batch files.

For example, here are some statements to test the lexer for numbers:

```
// Number tests
let i = 0;
i = 1.0;
i = 1.00;
i = 1.e1;
i = 1.E1;
i = 1.e+1;
i = 1.e-1;
i = 1.00e+1;
i = 1.00E+1;
i = 1e10;
i = 1e+1;
i = 1e-1;
```

There would be no output if all of them are recognized. Here's another set for strings. This time, they are printed out and the results saved.

```
// Strings
print "Strings and chars";
let s = "~!@#$%^&*()-+_=[]{}|;:'<>,./";
print s;
s = "0123456789 abcdefghijklmnopqrstuvwxyz";
print s;
s = " "" "" "" "" "" "";
print s;
print "";
```

The expected results:

```
Strings and chars
s = ~!@#$%^&*()-+_=[]{}|;:'<>,./
s = 0123456789 abcdefghijklmnopqrstuvwxyz
s =  " " " " " "
```

As the project moves on and more components have to interact with each other, the test cases become more involved – such as the following example for testing a `for` loop with flow control involved:

```
print "for loop within a for loop with continue statement";
for (i = 0; i < 4; i = i + 1)
{
      for (j = 0; j < 4; j = j + 1)
      {
            if (j == 2)
                  continue;
            print j;
```

```
      }
      if (i == 2)
            continue;
      print i;
      print "----";
}
```

The output from the above source, which is then used to compare with results generated from the latest version of the project:

```
for loop within a for loop with continue statement
j = 0
j = 1
j = 3
i = 0
----
j = 0
j = 1
j = 3
i = 1
----
j = 0
j = 1
j = 3
j = 0
j = 1
j = 3
i = 3
----
```

Test cases were generally written by the person who created the code. This is then reviewed by the other as a sanity check before adding it to the permanent test suite for regression testing.

Chapter 7: Lessons Learned

Chee Seng Choong:

- Start early and start small. Start small (no matter how small) to get some momentum going so that deadlines can be met. Large projects are always the bane of procrastinators. If you're one, go read the same points littered throughout the same chapters of the other projects to convince yourself.
- Consider using IDEs that integrate with some form of versioning system, such as Eclipse and CVS. It really helps to synchronize our work effortlessly.
- If you're using Eclipse, get the ANTLR plugin for it. It's very nice…

Thomas Chou:

- Displaying the Parse Tree in a UI component is very useful during debugging.
- Evaluate the 3rd party components carefully. Find out exactly what they can and can't do.
- Using a source control is a must even with if you only have a small group.

Appendix: Code Listing

grammar.g:

```
/* PCL lexer and parser
 */

class PCLLexer extends Lexer;

options{
       k = 2;
       charVocabulary = '\3'..'\377';
       testLiterals = false;
}

protected
ALPHA  : 'a'..'z' | 'A'..'Z' | '_';

protected
DIGIT  : '0'..'9';

ID options { testLiterals = true; }
             : ALPHA (ALPHA | DIGIT)* ;

NUMBER : (DIGIT)+ ('.' (DIGIT)*)? (('E'|'e') ('+'|'-')? (DIGIT)+)? ;

WS           : (' ' | '\t' | '\f')+
               { $setType(Token.SKIP); }
             ;

NL           : ("\r\n"                       // DOS/Windows
               | '\r'                        // Apple
               | '\n'                        // Unix
               )
               { newline(); $setType(Token.SKIP); }
             ;

COMMENT      : "//" (~( '\n' | '\r' ))* (NL)
               { $setType(Token.SKIP); }
             ;

STRING : '"'!
             (  ~('"' | '\n' )        // anything that don't match " or a
newline
          | ('"'!'"')                 // return " if it's preceeded by \
          )*
          '"'!
        ;

DOT                : '.';
LBRACE       : '{';
RBRACE       : '}';
LBRK         : '[';
RBRK         : ']';
LPAREN       : '(';
RPAREN       : ')';
SEMI         : ';';
COMMA        : ',';
```

```
PLUS         : '+';
MINUS        : '-';
MULT         : '*';
DIV                : '/';
MOD                : '%';
ASSIGN       : '=';
GT                 : '>';
LT                 : '<';

GE                 : ">=";
LE                 : "<=";
EQ                 : "==";
NEQ          : "!=";
NOT                : "!";
AND                : "&&";
PARALLEL     : "||";

class PCLParser extends Parser;

options {
       k = 4;
       buildAST = true;
}

tokens {
       STMT;
       NULL;
       ARRAY;
       LIST;
       FUNC_CALL;
       FUNC_DEF;
       GEN_DEF;
       VAL_DEF;
       STATEMENT;
}

program
       : (statement | func_def)* EOF!
         { #program = #([STMT, "program"], program); }
       ;

statement
       : for_stmt
    | if_stmt
    | return_stmt
    | break_stmt
    | continue_stmt
    | obj_ref_stmt
    | assign_stmt
    | let_stmt
       | print_stmt
    | LBRACE! (statement)* RBRACE!
      { #statement = #([STMT, "statement"], statement); }
    ;

for_stmt
       : "for"^ for_ctrl statement
       ;

for_ctrl
       : LPAREN! loop_init loop_cond loop_incr RPAREN!
       ;
```

```
loop_init
       : SEMI!      { #loop_init = #([NULL, "null_init"]); }
       | assign_stmt | let_stmt
       ;

loop_cond
       : SEMI!      { #loop_cond = #([NULL, "null_cond"]); }
       | relational_expr SEMI!
       ;

loop_incr
       : // empty
       { #loop_incr = #([NULL, "null_incr"]); }
       | ID ASSIGN^ expression
       ;

if_stmt
       : "if"^ LPAREN! expression RPAREN! statement
            (options {greedy = true;}: "else"! statement )?
       ;

return_stmt
       : "return"^ (expression)? SEMI!
       ;

break_stmt
       : "break"^ SEMI!
       ;

continue_stmt
       : "continue"^ SEMI!
       ;

assign_stmt
       : (obj_ref | array | ID) ASSIGN^ expression SEMI!
       ;

let_stmt
       : "let"^ ID ASSIGN! expression SEMI!
       ;

print_stmt
       : "print"^ (STRING | ID) SEMI!
       ;

array
       : ID LBRK! (expression) RBRK!
         { #array = #([ARRAY, "array"], array); }
       ;

list
       : expression (COMMA! expression)*
         { #list = #([LIST, "list"], list); }
       | /* empty */
         { #list = #([LIST, "empty_list"], list); }
       ;

obj_ref_stmt
       : ID (DOT^ func_call) SEMI!
       ;

obj_ref
       : ID (DOT^ (ID | array | func_call))+
```

```
        ;

func_call
        : ID LPAREN! list RPAREN!
          { #func_call = #([FUNC_CALL, "func_call"], func_call); }
        ;

func_def
        : LBRK! ID RBRK!
        ( "in"! LPAREN! list RPAREN! func_body
          { #func_def = #([GEN_DEF, "generate_def"], func_def); }
        | func_body
          { #func_def = #([VAL_DEF, "validate_def"], func_def); }
        )
        ;

func_body
        : LBRACE! (statement)+ RBRACE!
          { #func_body = #([STMT, "func_body"], func_body); }
        ;

expression
    : boolean_term ( "or"^ boolean_term )*
    ;

boolean_term
    : boolean_factor ( "and"^ boolean_factor )*
    ;

boolean_factor
    : ("not"^)? relational_expr
    ;

relational_expr
    : arith_expr ( (GE^ | LE^ | GT^ | LT^ | EQ^ | NEQ^) arith_expr )?
    ;

arith_expr
    : arith_term ( (PLUS^ | MINUS^) arith_term )*
    ;

arith_term
    : r_value ( (MULT^ | DIV^ | MOD^ ) r_value )*
    ;

r_value
        : ID
        | NUMBER
        | STRING
        | array
        | "true"
        | "false"
        | obj_ref
        | LPAREN! expression RPAREN!
        ;

cl_statement
        : ( statement | func_def )
    | "exit"
        { System.exit(0); }
    | EOF!
        { System.exit(0); }
    ;
```

walker.g

```
/*
 * walker.g : the AST walker.
 *
 * @author C.Choong
 */

{
import java.io.*;
import java.util.*;
}

class PCLAntlrWalker extends TreeParser;
options{
       importVocab = PCLLexer;
}

{
    static PCLDataType null_data = new PCLDataType( "<NULL>" );
    PCLInterpreter ipt = PCL.ipt;
    PCLUniqueID uid = new PCLUniqueID();
}

expr returns [ PCLDataType r ]
{
    PCLDataType a, b;
    Vector v;
    PCLDataType[] x;
    String s = null;
    String[] sx;
    r = null_data;
}
             : #(DOT a=expr right_op:.)
                    {
                           ipt.currentObj = a;
                           r = expr(#right_op);
                    }
        | #("or" a=expr right_or:.)
            {
                if ( a instanceof PCLBool )
                    r = ( ((PCLBool)a).var ? a : expr(#right_or) );
                else
                    r = a.or( expr(#right_or) );
            }
        | #("and" a=expr right_and:.)
            {
                if ( a instanceof PCLBool )
                    r = ( ((PCLBool)a).var ? expr(#right_and) : a );
                else
                    r = a.and( expr(#right_and) );
            }
        | #("let" varname:ID b=expr) { r = ipt.setVariable( varname.getText(),
b); }
             | #("print" a=expr)               { a.print(); }
        | #("not" a=expr)            { r = a.not(); }
        | #(GE a=expr b=expr)        { r = a.ge( b ); }
        | #(LE a=expr b=expr)        { r = a.le( b ); }
        | #(GT a=expr b=expr)        { r = a.gt( b ); }
        | #(LT a=expr b=expr)        { r = a.lt( b ); }
        | #(EQ a=expr b=expr)        { r = a.eq( b ); }
```

```
        | #(NEQ a=expr b=expr)       { r = a.ne( b ); }
        | #(PLUS a=expr b=expr)      { r = a.plus( b ); }
        | #(MINUS a=expr b=expr)     { r = a.minus( b ); }
        | #(MULT a=expr b=expr)      { r = a.times( b ); }
        | #(DIV a=expr b=expr)           { r = a.div(b); }
        | #(MOD a=expr b=expr)       { r = a.modulus( b ); }
        | #(ASSIGN a=expr b=expr)    { r = ipt.assign( a, b ); }
        | #(ARRAY a=expr index:.) { r = ipt.accessCollection(a, expr(#index));
}
        | #(FUNC_CALL a=expr x=mexpr)
          { r = ipt.funcInvoke( this, a, x ); }
        | num:NUMBER                 { r = ipt.getNumber( num.getText() ); }
        | str:STRING                 { r = new PCLString( str.getText() ); }
        | "true"                     { r = new PCLBool( true ); }
        | "false"                    { r = new PCLBool( false ); }
        | id:ID                          { r = ipt.getVariable( id.getText() );
}
        | empty:NULL                              { r = new PCLBool( true );  }
        | #("for" a=expr for_cond:. for_incr:. for_body:.)
           {  long i = uid.get();
                    b = expr(#for_cond);
                    if ( !(b instanceof PCLBool)) {
                           if (b == null) {
                                  b = new PCLBool( true );
                           }
                           else {
                                  return b.error("for_cond must return a boolean
type");
                           }
                    }
                 while ( ((PCLBool)b).var && ipt.canProceed()) {
                           r = expr(#for_body);
                           ipt.loopNext( String.valueOf(i) );
                           r = expr(#for_incr);
                           b = expr(#for_cond);
                    }
                    ipt.loopEnd( String.valueOf(i) );
           }
        | #("if" a=expr thenp:. (elsep:.)?)
            {
                if ( !( a instanceof PCLBool ) )
                    return a.error( "if: expression must return a boolean type"
);
                if ( ((PCLBool)a).var )
                    r = expr( #thenp );
                else if ( null != elsep )
                    r = expr( #elsep );
            }
        | #(STMT (stmt:. { if ( ipt.canProceed() ) r = expr(#stmt); } )*)
        | #("break" (breakid:ID      { s = breakid.getText(); }
                    )?
           )                         { ipt.setBreak( s ); }
        | #("continue" (contid:ID    { s = contid.getText(); }
                       )?
           )                         { ipt.setContinue( s ); }
        | #("return" ( a=expr        { r = ipt.rvalue( a ); }
                     )?
           )                         { ipt.setReturn( null ); }
        | #(GEN_DEF gname:ID sx=vlist gbody:.)
             { ipt.generatorRegister( gname.getText(), sx, #gbody); }
             | #(VAL_DEF vname:ID vbody:.)
                    { ipt.validatorRegister( vname.getText(), #vbody); }
        ;
```

```
mexpr returns [ PCLDataType[] rv ]
{
    PCLDataType a;
    rv = null;
    Vector v;
}
        : #(LIST         { v = new Vector(); }
                ( a=expr      { v.add( a ); }
                )*
           )                 { rv = ipt.convertExprList( v ); }
        | a=expr             { rv = new PCLDataType[1]; rv[0] = a; }
        ;

vlist returns [ String[] sv ]
{
    Vector v;
    sv = null;
}
        : #(LIST     { v = new Vector(); }
             (s:ID       { v.add( s.getText() ); }
             )*
          )              { sv = ipt.convertVarList( v ); }
        ;
```

PCLMain.java

```
import java.io.*;
import antlr.CommonAST;
import antlr.collections.AST;
import antlr.debug.misc.ASTFrame;

/*
 * @author C.Choong
 */
public class PCLMain
{
    static boolean showtree = false;

    public static void main(String[] args) {

        showtree = args.length >=1 && args[0].equals("-t");

        if ( args.length >= 1 && args[args.length-1].charAt(0) != '-' )
            execFile( args[args.length-1] );
        else
            interpreter();
    }

    public static void execFile( String filename ) {
        try
        {
            InputStream input = ( null != filename ) ?
                (InputStream) new FileInputStream( filename ) :
                (InputStream) System.in;

            PCLLexer lexer = new PCLLexer( input );
            PCLParser parser = new PCLParser( lexer );

            // Parse the input program
            parser.program();

            // Get AST from the parser
            CommonAST tree = (CommonAST)parser.getAST();

            // Open a window in which the AST is displayed graphically
            if (showtree) {
                ASTFrame frame = new ASTFrame("AST from the Simp parser",
tree);
                frame.setVisible(true);
            }

            PCLAntlrWalker walker = new PCLAntlrWalker();

            // Traverse the tree created by the parser
            PCLDataType r = walker.expr( tree );
        } catch( IOException e ) {
            System.err.println( "Error: I/O: " + e );
        } catch( Exception e ) {
            System.err.println( "Error: " + e );
        }
    }


    public static void interpreter() {
        InputStream input = (InputStream) new DataInputStream( System.in );
```

```
        PCLAntlrWalker walker = new PCLAntlrWalker();

        while(true)
        {
            try {
                while( input.available() > 0 )
                    input.read();
            }
            catch ( IOException e ) {}

            System.out.print( "Pcl> " );
            System.out.flush();

            PCLLexer lexer = new PCLLexer( input );
            PCLParser parser = new PCLParser( lexer );

            try {
                parser.cl_statement();
                CommonAST tree = (CommonAST)parser.getAST();
                PCLDataType r = walker.expr(tree);
                if (r.name != "<NULL>")
                    r.print();
            } catch(Exception e) {
                System.err.println("Recognition exception " + e);
            }
        }
    }
}
```

CSV.java

```
/*
 * Copyright (c) Ian F. Darwin, http://www.darwinsys.com/, 1996-2002.
 * All rights reserved. Software written by Ian F. Darwin and others.
 * $Id: CSV.java,v 1.2 2006/05/07 19:40:52 cvs Exp $
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 *
 * THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS''
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
 * TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
 * PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS
 * BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 *
 * Java, the Duke mascot, and all variants of Sun's Java "steaming coffee
 * cup" logo are trademarks of Sun Microsystems. Sun's, and James Gosling's,
 * pioneering role in inventing and promulgating (and standardizing) the Java
 * language and environment is gratefully acknowledged.
 *
 * The pioneering role of Dennis Ritchie and Bjarne Stroustrup, of AT&T, for
 * inventing predecessor languages C and C++ is also gratefully acknowledged.
 */
import java.util.*;

/** Parse comma-separated values (CSV), a common Windows file format.
 * Sample input: "LU",86.25,"11/4/1998","2:19PM",+4.0625
 * <p>
 * Inner logic adapted from a C++ original that was
 * Copyright (C) 1999 Lucent Technologies
 * Excerpted from 'The Practice of Programming'
 * by Brian W. Kernighan and Rob Pike.
 * <p>
 * Included by permission of the http://tpop.awl.com/ web site,
 * which says:
 * "You may use this code for any purpose, as long as you leave
 * the copyright notice and book citation attached." I have done so.
 * @author Brian W. Kernighan and Rob Pike (C++ original)
 * @author Ian F. Darwin (translation into Java and removal of I/O)
 * @author Ben Ballard (rewrote advQuoted to handle '""' and for readability)
 */
public class CSV {

  public static final char DEFAULT_SEP = ',';

  /** Construct a CSV parser, with the default separator (`,'). */
  public CSV() {
    this(DEFAULT_SEP);
```

```
  }

  /** Construct a CSV parser with a given separator.
   * @param sep The single char for the separator (not a list of
   * separator characters)
   */
  public CSV(char sep) {
    fieldSep = sep;
  }

  /** The fields in the current String */
  protected List list = new ArrayList();

  /** the separator char for this parser */
  protected char fieldSep;

  /** parse: break the input String into fields
   * @return java.util.Iterator containing each field
   * from the original as a String, in order.
   */
  @SuppressWarnings("unchecked")
public List parse(String line)
  {
    StringBuffer sb = new StringBuffer();
    list.clear();      // recycle to initial state
    int i = 0;

    if (line.length() == 0) {
      list.add(line);
      return list;
    }

    do {
            sb.setLength(0);
            if (i < line.length() && line.charAt(i) == '"')
                i = advQuoted(line, sb, ++i);  // skip quote
            else
                i = advPlain(line, sb, i);
            list.add(sb.toString());
      i++;
    } while (i < line.length());

    return list;
  }

  /** advQuoted: quoted field; return index of next separator */
  protected int advQuoted(String s, StringBuffer sb, int i)
  {
    int j;
    int len= s.length();
        for (j=i; j<len; j++) {
            if (s.charAt(j) == '"' && j+1 < len) {
                if (s.charAt(j+1) == '"') {
                    j++; // skip escape char
                } else if (s.charAt(j+1) == fieldSep) { //next delimeter
                    j++; // skip end quotes
                    break;
                }
            } else if (s.charAt(j) == '"' && j+1 == len) { // end quotes at end
of line
                break; //done
      }
      sb.append(s.charAt(j));  // regular character.
```

```
    }
    return j;
  }

  /** advPlain: unquoted field; return index of next separator */
  protected int advPlain(String s, StringBuffer sb, int i)
  {
    int j;

    j = s.indexOf(fieldSep, i); // look for separator
        if (j == -1) {                // none found
            sb.append(s.substring(i));
            return s.length();
        } else {
            sb.append(s.substring(i, j));
            return j;
        }
    }
}
```

PCL.java

```
/*
 * @author: T. Chou
 */
public class PCL
{
       public static PCLChart chart = new PCLChart();
       public static PCLInterpreter ipt = new PCLInterpreter();
       public static PCLAntlrWalker walker = new PCLAntlrWalker();

}
```

PCLBool.java

```
import java.io.PrintWriter;

/**
 * the wrapper class for boolean
 * @author Hanhua
 *
 * Modified by T. Chou
 */
class PCLBool extends PCLDataType {
    boolean var;

    PCLBool( boolean var ) {
        this.var = var;
    }

    public String typename() {
        return "bool";
    }

    public PCLDataType copy() {
        return new PCLBool( var );
    }

    public void print( PrintWriter w ) {
        if ( name != null )
            w.print( name + " = " );
        w.println( var ? "true" : "false" );
    }


    public PCLDataType and( PCLDataType b ) {
        if ( b instanceof PCLBool )
            return new PCLBool( var && ((PCLBool)b).var );
        return error( b, "and" );
    }

    public PCLDataType or( PCLDataType b ) {
        if ( b instanceof PCLBool )
            return new PCLBool( var || ((PCLBool)b).var );
        return error( b, "or" );
    }

    public PCLDataType not() {
        return new PCLBool( !var );
    }

    public PCLDataType eq( PCLDataType b ) {
        // not exclusive or
        if ( b instanceof PCLBool )
            return new PCLBool( ( var && ((PCLBool)b).var )
                                || ( !var && !((PCLBool)b).var ) );
        return error( b, "==" );
    }

    public PCLDataType ne( PCLDataType b ) {
        // exclusive or
        if ( b instanceof PCLBool )
            return new PCLBool( ( var && !((PCLBool)b).var )
                                || ( !var && ((PCLBool)b).var ) );
```

```
            return error( b, "!=" );
        }
    }
```

PCLChart.java

```
import javax.swing.JPanel;
import java.io.*;
import java.util.*;

import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartPanel;
import org.jfree.chart.ChartFrame;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.axis.NumberAxis;

import org.jfree.chart.plot.PlotOrientation;
import org.jfree.chart.plot.XYPlot;
import org.jfree.chart.renderer.xy.*;
import org.jfree.data.xy.XYDataset;
import org.jfree.data.xy.XYSeries;
import org.jfree.data.xy.XYSeriesCollection;
import org.jfree.ui.ApplicationFrame;
import org.jfree.ui.RectangleInsets;
import org.jfree.ui.RefineryUtilities;

import antlr.RecognitionException;

/*
 * @author T. Chou
 */
public class PCLChart extends PCLDataType
{
	// The chart title - name of the CSV file
	public PCLString title = new PCLString("");

	// A Hashtable of DataColumns
	// key: name of the data column
	// value: DataColumn object
	public HashMap<String, PCLDataColumn> _datacolumns = new HashMap<String,
PCLDataColumn>();

	// A wrapper around the datacolumns collection
	public PCLCollectionContainer datacolumns = new
PCLCollectionContainer(_datacolumns);

	// A Hashtable of DataSeries objects
	// key: name of DataSeries
	// value: DataSeries object
	public HashMap<String, PCLDataSeries> _dataseries = new HashMap<String,
PCLDataSeries>();

	// A wrapper around the dataseries collection
	public PCLCollectionContainer dataseries = new
PCLCollectionContainer(_dataseries);

	// The one and only chart control
	protected JFreeChart chart;

	// Window
	protected ChartFrame frame;

	public PCLChart()
	{
		this.name = "PCLChart";
```

```
            this.title = new PCLString("");
      }

      // Called by Interpreter to validate the data for a specific column in
the CSV file
    public PCLBool validate(PCLValidator validator, PCLString colName) throws
RecognitionException
    {
      PCLDataColumn col = _datacolumns.get(colName.var);
      if(col == null)
      {
            throw new PCLException("Column "+ colName.var + " does not
exist");
      }

      boolean success = true;

      // Loop through all the elements of the column collection and call
      // the validation code defined by the user
      for(int i=0; i<col._data.size(); i++)
      {
            PCLDouble value = col._data.get(i);
            validator.setValue(value);
            PCLDataType result = PCL.ipt.invokePCLFunc(validator, new
PCLDataType[0]);
            if(result instanceof PCLBool && ((PCLBool)result).var)
            {
                  col._data.set(i, new PCLDouble(validator.getValue().var));
            }
            else
            {
                  System.out.println("Validation failed at location: "+ i + "
with data value: "+value.var);
                  success = false;
            }

            validator.resetSymbolTable();
      }

      if(success)
            System.out.println("Valdation: " + validator.name + " succeeded");
      return new PCLBool(success);
    }

    // Generate a new data series
    public PCLBool generate(PCLGenerator generator, PCLString seriesname,
PCLString type, PCLDataType args[]) throws RecognitionException
    {
            if(_dataseries.containsKey(seriesname.var))
            {
                  throw new PCLException("Unable to add new series: "+name+"
\r\n DataSeries name already used");
            }

            PCLDataSeries newseries = new PCLDataSeries();
            newseries.id.var = seriesname.var;
            newseries.type = type.var;

            generator.setDataSeries(newseries);

            PCLDataType result = PCL.ipt.invokePCLFunc(generator, args);

            if(result instanceof PCLBool && ((PCLBool)result).var)
```

```
            {
                  newseries = generator.getDataSeries();
                  _dataseries.put(newseries.id.var, newseries);
                  generator.resetSymbolTable();
            return new PCLBool(true);
            }

            return new PCLBool(false);
    }

    // Initialize the chart with data from a CSV file
       public PCLBool load(PCLString file)
       {
              // Clear data
              _datacolumns.clear();
              _dataseries.clear();

              CSV csv = new CSV();

              BufferedReader is = null;
              int rowCount = 0;
              try
              {
                     // open file for reading
                     is = new BufferedReader(new FileReader(file.var));

                  String line;
                  boolean firstLine = false;
                  ArrayList<String> colNames = new ArrayList<String>();
                  while ((line = is.readLine()) != null)
                  {
                     rowCount++;
                         Iterator e = csv.parse(line).iterator();

                         // First line is always the column names
                     if(firstLine == false)
                     {
                            // Get the column names and create a DataColumn
object for each entry
                            while (e.hasNext())
                            {
                                   PCLDataColumn dataColumn = new
PCLDataColumn();
                                   dataColumn.id.var = (String)e.next();

       if(_datacolumns.containsKey(dataColumn.id.var))
                                   {
                                          throw new PCLException("Duplicate
Column name found: "+dataColumn.name+" \r\n Aborting initialization");
                                   }
                                   _datacolumns.put(dataColumn.id.var,
dataColumn);
                                   colNames.add(dataColumn.id.var);
                            }

                            firstLine = true;
                     }
                     else
                     {
                            // Now populate the DataColumn objects with data
                            int i = 0;
                            while (e.hasNext())
                            {
```

```
                                    PCLDataColumn col =
_datacolumns.get(colNames.get(i));
                                    try
                                    {
                                          col._data.add(new
PCLDouble(Double.parseDouble((String)e.next())));
                                    }
                                    catch(NumberFormatException ne)
                                    {
                                          throw new PCLException("Non-numeric
encountered in CSV file. Aborting");
                                    }
                                    i++;
                              }

                              if(i != _datacolumns.size())
                              {
                                    throw new PCLException("Bad CSV file. Not all
columns have the same number of data points. Aborting");
                              }
                        }
                    }
              }
              catch(IOException e)
              {
                    throw new PCLException(e.toString());
              }

              this.title = new PCLString(file.var);
              return new PCLBool(true);
        }

        // Adds a new data series to the chart object
        public PCLBool addseries(PCLString name, PCLString xCol, PCLString yCol,
PCLString type)
        {
              if(_dataseries.containsKey(name.var))
              {
                    throw new PCLException("Unable to add new series: "+name+"
\r\n DataSeries name already used");
              }

              PCLDataSeries d = new PCLDataSeries();
              d.id.var = name.var;
              d._x.addAll(_datacolumns.get(xCol.var)._data);
              d._y.addAll(_datacolumns.get(yCol.var)._data);
              String plotType = type.var;
              if(plotType.compareToIgnoreCase("scatter")==0 ||
                          plotType.compareToIgnoreCase("line") == 0 ||
                          plotType.compareToIgnoreCase("bar") == 0)
                    d.type = type.var;
              else
              {
                    throw new PCLException("Invalid DataSeries type
specified");
              }
              _dataseries.put(name.var, d);
              return new PCLBool(true);
        }

        protected XYSeriesCollection scatterDS = new XYSeriesCollection();
        protected XYSeriesCollection lineDS = new XYSeriesCollection();
        protected XYSeriesCollection barDS = new XYSeriesCollection();
```

```
	// Generates a DataSet to be used by JFreeChart for plotting
	// This is where all the DataSeries objects are created and populated
	protected void generateDataSet()
	{
		scatterDS = new XYSeriesCollection();
		lineDS = new XYSeriesCollection();
		barDS = new XYSeriesCollection();

		Set<String> names = _dataseries.keySet();

		for (Iterator it = names.iterator(); it.hasNext(); )
		{
			// Name the series
			String name = (String)it.next();
			XYSeries series = new XYSeries(name);
			series.setDescription(name);

			// Populate the data
			PCLDataSeries data = _dataseries.get(name);
			for(int i=0; i<data._x.size(); i++)
			{
				series.add(data._x.get(i).var, data._y.get(i).var);
			}
			// Add the series to the chart's DataSet
			if(data.type.compareToIgnoreCase("Scatter")==0)
				scatterDS.addSeries(series);
			else if(data.type.compareToIgnoreCase("Line")==0)
				lineDS.addSeries(series);
			else if(data.type.compareToIgnoreCase("Bar")==0)
				barDS.addSeries(series);
		}
	}

	// Plot the chart
	public PCLBool plot()
	{
		if(frame != null)
			frame.setVisible(false);

		// Generate chart data set
		generateDataSet();

		//	create the chart...
		chart = ChartFactory.createXYLineChart(
				title.var, // chart title
				"", // x axis label
				"", // y axis label
				new XYSeriesCollection(), // data
				PlotOrientation.VERTICAL,
				true, // include legend
				true, // tooltips
				false // urls
		);

		XYPlot plot = chart.getXYPlot();

		// Scatter Plot Renderer
		plot.setDataset(1, scatterDS);
		StandardXYItemRenderer renderer = new
StandardXYItemRenderer(StandardXYItemRenderer.SHAPES);
		renderer.setPlotLines(false);
		plot.setRenderer(1, renderer);
```

```
            // Line Plot renderer
            plot.setDataset(2, lineDS);
            renderer = new
StandardXYItemRenderer(StandardXYItemRenderer.LINES);
            renderer.setPlotLines(true);
            plot.setRenderer(2, renderer);

            // Bar Plot renderer
            plot.setDataset(3, barDS);
            XYBarRenderer barRenderer = new XYBarRenderer();
            plot.setRenderer(3, barRenderer);

            frame = new ChartFrame("PCL Chart", chart);
            frame.pack();
            frame.setLocation(100, 100);
            frame.setVisible(true);

            return new PCLBool(true);
      }

    public void print( PrintWriter w )
    {
        if ( name != null )
            w.println( name + " = "+ "PCLChart");
    }

    public PCLDataType copy() {
        return this;
    }
}

// The DataSeries object
class PCLDataSeries extends PCLDataType
{
      // Name of the DataSeries
      public PCLString id = new PCLString("");

      // X-values
      public ArrayList<PCLDouble> _x = new ArrayList<PCLDouble>();

      public PCLCollectionContainer x = new PCLCollectionContainer(_x);

      // Y-values
      public ArrayList<PCLDouble> _y = new ArrayList<PCLDouble>();

      public PCLCollectionContainer y = new PCLCollectionContainer(_y);

      public String type = "Scatter";

      public PCLBool add(PCLDouble xNum, PCLDouble yNum)
      {
      _x.add(new PCLDouble(xNum.var));
      _y.add(new PCLDouble(yNum.var));

      return new PCLBool(true);
      }

      public PCLBool add(PCLInt xNum, PCLDouble yNum)
      {
      _x.add(new PCLDouble(xNum.var));
      _y.add(new PCLDouble(yNum.var));
```

```
        return new PCLBool(true);
        }

        public PCLBool add(PCLDouble xNum, PCLInt yNum)
        {
        _x.add(new PCLDouble(xNum.var));
        _y.add(new PCLDouble(yNum.var));

        return new PCLBool(true);
        }

        public PCLBool add(PCLInt xNum, PCLInt yNum)
        {
        _x.add(new PCLDouble(xNum.var));
        _y.add(new PCLDouble(yNum.var));

        return new PCLBool(true);
        }

    public void print( PrintWriter w )
    {
        if ( name != null )
            w.println( name + " = "+ "PCLDataSeries contents: ");
        String val = "x = ";
              for(int i=0; i<_x.size(); i++)
              {
                    val += _x.get(i).var+" ";
              }
        w.println( val );
        val = "y = ";
              for(int i=0; i<_y.size(); i++)
              {
                    val += _y.get(i).var+" ";
              }
        w.println( val );

    }

    public PCLDataType copy() {
        return this;
    }

}

// The DataColumn object
class PCLDataColumn extends PCLDataType
{
       public PCLDataColumn()
       {
             this.name = "PCLDataColumn";
       }
       // Name of the DataColumn
       public PCLString id = new PCLString("");

       // Data for this column
       public ArrayList<PCLDouble> _data = new ArrayList<PCLDouble>();

       public PCLCollectionContainer data = new PCLCollectionContainer(_data);

    public void print( PrintWriter w )
    {
        if ( name != null )
            w.print( name + " = " + "PCLDataColumn contents: " );
```

```
        String val = "";
             for(int i=0; i<_data.size(); i++)
             {
                    val += _data.get(i).var+" ";
             }
        w.println( val );
    }

    public PCLDataType copy() {
        return this;
    }
}
```

PCLCollection.java

```
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.HashMap;

/*
 * @author T.Chou
 */
public interface PCLCollection
{
	PCLDataType get(PCLDataType key);

	PCLInt length();
}

// Used by Interpreter to retrieve items from chart collections
class PCLCollectionContainer extends PCLDataType
	implements PCLCollection
{
	protected Object collection;

	public PCLCollectionContainer(Object collection)
	{
		this.name = "PCLCollectionContainer";
		this.collection = collection;
	}

	public PCLDataType get(PCLDataType key)
	{
		if(collection instanceof HashMap)
		{
			HashMap map = (HashMap)collection;
			if(key instanceof PCLString)
			{
				PCLString keyStr = (PCLString)key;
				if(map.containsKey(keyStr.var))
					return (PCLDataType)map.get(keyStr.var);
				else
					throw new PCLException("Unable to locate item
in collection");
			}
			else
				throw new PCLException("Access to this collection is
by String only");
		}
		else if(collection instanceof ArrayList)
		{
			ArrayList list = (ArrayList)collection;
			if(key instanceof PCLInt)
			{
				PCLInt keyInt = (PCLInt)key;
				return (PCLDataType)list.get(keyInt.var);
			}
			else
				throw new PCLException("Access to this collection is
by integer only");
		}

		throw new PCLException("Unable to access collection. Unknown
collection type");
	}
```

```
	public PCLInt length()
	{
		if(collection instanceof HashMap)
		{
			HashMap map = (HashMap)collection;
			return new PCLInt(map.size());
		}
		else if(collection instanceof ArrayList)
		{
			ArrayList list = (ArrayList)collection;
			return new PCLInt(list.size());
		}

		throw new PCLException("Unable to access collection length info.
Unknown collection type");
	}

    public PCLDataType copy() {
        return this;
    }

    public void print( PrintWriter w )
    {
        if ( name != null )
            w.println( name + " = " + "PCLCollection " + "- size:
"+length().var);
    }
}
```

PCLDataType.java

```
import java.io.PrintWriter;

/**
 * The base data type class (also a meta class)
 *
 * Error messages are generated here.
 * @author Hanhua
 *
 * modified by T.Chou
 */
public class PCLDataType
{
    public String name;   // used in hash table

    public PCLDataType() {
        name = null;
    }

    public PCLDataType( String name ) {
        this.name = name;
    }

    public PCLDataType isTypeOf(PCLString regex)
    {
       return error( "isTypeOf not supported for this data type" );
    }

    public String typename() {
        return "unknown";
    }

    public PCLDataType copy() {
        return new PCLDataType();
    }

    public void setName( String name ) {
        this.name = name;
    }

    public PCLDataType error( String msg ) {
        throw new PCLException( "illegal operation: " + msg
                                + "( <" + typename() + "> "
                                + ( name != null ? name : "<?>" )
                                + " )" );
    }

    public PCLDataType error( PCLDataType b, String msg ) {
        if ( null == b )
            return error( msg );
        throw new PCLException(
            "illegal operation: " + msg
            + "( <" + typename() + "> "
            + ( name != null ? name : "<?>" )
            + " and "
            + "<" + typename() + "> "
            + ( name != null ? name : "<?>" )
            + " )" );
    }

    public void print( PrintWriter w ) {
```

```
    if ( name != null )
        w.print( name + " = " );
    w.println( "<undefined>" );
}

public void print() {
    print( new PrintWriter( System.out, true ) );
}

public PCLDataType assign( PCLDataType b ) {
    return error( b, "=" );
}

public PCLDataType uminus() {
    return error( "-" );
}

public PCLDataType plus( PCLDataType b ) {
    return error( b, "+" );
}

public PCLDataType add( PCLDataType b ) {
    return error( b, "+=" );
}

public PCLDataType minus( PCLDataType b ) {
    return error( b, "-" );
}

public PCLDataType sub( PCLDataType b ) {
    return error( b, "-=" );
}

public PCLDataType times( PCLDataType b ) {
    return error( b, "*" );
}

public PCLDataType mul( PCLDataType b ) {
    return error( b, "*=" );
}

public PCLDataType lfracts( PCLDataType b ) {
    return error( b, "/" );
}

public PCLDataType rfracts( PCLDataType b ) {
    return error( b, "/\'" );
}

public PCLDataType div( PCLDataType b) {
    return error ( b, "/" );
}

public PCLDataType modulus( PCLDataType b ) {
    return error( b, "%" );
}

public PCLDataType rem( PCLDataType b ) {
    return error( b, "%=" );
}

public PCLDataType gt( PCLDataType b ) {
    return error( b, ">" );
```

```
    }

    public PCLDataType ge( PCLDataType b ) {
        return error( b, ">=" );
    }

    public PCLDataType lt( PCLDataType b ) {
        return error( b, "<" );
    }

    public PCLDataType le( PCLDataType b ) {
        return error( b, "<=" );
    }

    public PCLDataType eq( PCLDataType b ) {
        return error( b, "==" );
    }

    public PCLDataType ne( PCLDataType b ) {
        return error( b, "!=" );
    }

    public PCLDataType and( PCLDataType b ) {
        return error( b, "and" );
    }

    public PCLDataType or( PCLDataType b ) {
        return error( b, "or" );
    }

    public PCLDataType not() {
        return error( "not" );
    }
}
```

PCLDouble.java

```
import java.io.PrintWriter;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

/**
 * The wrapper class for double
 *
 * @author Hanhua
 *
 * Modified by T.Chou
 */
class PCLDouble extends PCLDataType
       implements Comparable
{
    double var;

    public PCLDouble( double x ) {
        var = x;
    }

    public String typename() {
        return "double";
    }

    public PCLDataType copy() {
        return new PCLDouble( var );
    }

    public static double doubleValue( PCLDataType b ) {
        if ( b instanceof PCLDouble )
            return ((PCLDouble)b).var;
        if ( b instanceof PCLInt )
            return (double) ((PCLInt)b).var;
        b.error( "cast to double" );
        return 0;
    }

    public void print( PrintWriter w ) {
        if ( name != null )
            w.print( name + " = " );
        w.println( Double.toString( var ) );
    }

    public PCLDataType uminus() {
        return new PCLDouble( -var );
    }

    public PCLDataType plus( PCLDataType b ) {
        return new PCLDouble( var + doubleValue(b) );
    }

    public PCLDataType add( PCLDataType b ) {
        var += doubleValue( b );
        return this;
    }

    public PCLDataType minus( PCLDataType b ) {
        return new PCLDouble( var - doubleValue(b) );
    }
```

```
public PCLDataType sub( PCLDataType b ) {
    var -= doubleValue( b );
    return this;
}

public PCLDataType times( PCLDataType b ) {
    return new PCLDouble( var * doubleValue(b) );
}

public PCLDataType mul( PCLDataType b ) {
    var *= doubleValue( b );
    return this;
}

public PCLDataType lfracts( PCLDataType b ) {

    return new PCLDouble( var / doubleValue(b) );
}

public PCLDataType rfracts( PCLDataType b ) {
    return lfracts( b );
}

public PCLDataType div( PCLDataType b) {
    return new PCLDouble( var / doubleValue(b) );
}

public PCLDataType modulus( PCLDataType b ) {
    return new PCLDouble( var % doubleValue(b) );
}


public PCLDataType rem( PCLDataType b ) {
    var %= doubleValue( b );
    return this;
}

public PCLDataType gt( PCLDataType b ) {
    return new PCLBool( var > doubleValue(b) );
}

public PCLDataType ge( PCLDataType b ) {
    return new PCLBool( var >= doubleValue(b) );
}

public PCLDataType lt( PCLDataType b ) {
    return new PCLBool( var < doubleValue(b) );
}

public PCLDataType le( PCLDataType b ) {
    return new PCLBool( var <= doubleValue(b) );
}

public PCLDataType eq( PCLDataType b ) {
    return new PCLBool( var == doubleValue(b) );
}

public PCLDataType ne( PCLDataType b ) {
    return new PCLBool( var != doubleValue(b) );
}

public boolean equals(Object arg0)
{
```

```
       return compareTo(arg0) == 0;
    }

       public int compareTo(Object arg0) {
             return new Double(this.var).compareTo(new
Double(((PCLDouble)arg0).var));
       }

       public PCLDataType isTypeOf(PCLString regex)
       {
        Pattern pattern = Pattern.compile(regex.var);
        Matcher matcher = pattern.matcher(""+this.var);

        return new PCLBool(matcher.find());
       }
}
```

PCLException.java

```
/**
 * Exception class: messages are generated in various classes
 * @author Hanhua
 */
class PCLException extends RuntimeException {
       /**
        *
        */
       private static final long serialVersionUID = 1L;

       PCLException( String msg ) {
         System.err.println( "Error: " + msg );
    }
}
```

PCLFunction.java

```
import java.io.PrintWriter;
import java.util.HashMap;

import antlr.collections.AST;

/**
 * Technically, PCL does not have functions
 * PCLValidator and PCLGenerator are the closest thing to functions and
 * will use this thing as a base class
 * Can also represent a function declared in one of the charting objects
 *
 * @author T.Chou
 */
class PCLFunction extends PCLDataType
{
//      Function arguments
    String[] args;

    //  body = null means chart object function.
    AST body;

//  the symbol table for this function
    HashMap<String, PCLDataType> pst;

    // the object that has this function as a member variable
    PCLDataType owner;

    public PCLFunction( String name, String[] args, AST body) {
        super( name );
        this.args = args;
        this.body = body;
        pst = new HashMap<String, PCLDataType>();
        pst.put("chart", PCL.chart);
    }

    public PCLFunction( String name, int id, PCLDataType owner ) {
        super( name );
        this.args = null;
        pst = null;
        body = null;
        this.owner = owner;
    }

    public final boolean isInternal() {
        return body == null;
    }

    public String typename() {
        return "function";
    }

    public PCLDataType copy() {
        return new PCLFunction( name, args, body );
    }

    public void print( PrintWriter w ) {
        if ( body == null )
        {
            w.println( name );
        }
```

```
        else
        {
            if ( name != null )
                w.print( name + " = " );
            w.print( "<function>(" );
            for ( int i=0; ; i++ )
            {
                w.print( args[i] );
                if ( i >= args.length - 1 )
                    break;
                w.print( "," );
            }
            w.println( ")" );
        }
    }

    public String[] getArgs() {
        return args;
    }

    public HashMap<String, PCLDataType> getSymbolTable() {
        return pst;
    }

    public AST getBody() {
        return body;
    }

    public void resetSymbolTable() {
       pst.clear();
        pst.put("chart", PCL.chart);
    }
}
```

PCLGenerator.java

```
import antlr.collections.AST;

/*
 * The Generator object used to create new data series
 * @author T.Chou
 */
public class PCLGenerator extends PCLFunction
{
    public PCLGenerator( String name, String[] args, AST body)
    {
       super(name, args, body);
    }

    public void setDataSeries(PCLDataSeries series)
    {
       pst.put("newseries", series);
    }

    public PCLDataSeries getDataSeries()
    {
       PCLDataType retval = pst.get("newseries");
       if(retval instanceof PCLDataSeries)
              return (PCLDataSeries)retval;
       else
              throw new PCLException("DataSeries object has been corrupted
during generation. Aborting transaction");
    }
}
```

PCLInt.java

```
import java.io.PrintWriter;
import java.util.regex.*;

/**
 * the wrapper class of int
 *
 * @author Hanhua
 *
 * modified by T.Chou
 */
class PCLInt extends PCLDataType
       implements Comparable
{
    int var;

    public PCLInt( int x ) {
        var = x;
    }

    public String typename() {
        return "int";
    }

    public PCLDataType copy() {
        return new PCLInt( var );
    }

    public static int intValue( PCLDataType b ) {
        if ( b instanceof PCLDouble )
            return (int) ((PCLDouble)b).var;
        if ( b instanceof PCLInt )
            return ((PCLInt)b).var;
        b.error( "cast to int" );
        return 0;
    }

    public void print( PrintWriter w ) {
        if ( name != null )
            w.print( name + " = " );
        w.println( Integer.toString( var ) );
    }

    public PCLDataType uminus() {
        return new PCLInt( -var );
    }

    public PCLDataType plus( PCLDataType b ) {
        if ( b instanceof PCLInt )
            return new PCLInt( var + intValue(b) );
        return new PCLDouble( var + PCLDouble.doubleValue(b) );
    }

    public PCLDataType add( PCLDataType b ) {
        var += intValue( b );
        return this;
    }

    public PCLDataType minus( PCLDataType b ) {
        if ( b instanceof PCLInt )
            return new PCLInt( var - intValue(b) );
```

```
    return new PCLDouble( var - PCLDouble.doubleValue(b) );
}

public PCLDataType sub( PCLDataType b ) {
    var -= intValue( b );
    return this;
}

public PCLDataType times( PCLDataType b ) {
    if ( b instanceof PCLInt )
        return new PCLInt( var * intValue(b) );
    return new PCLDouble( var * PCLDouble.doubleValue(b) );
}

public PCLDataType mul( PCLDataType b ) {
    var *= intValue( b );
    return this;
}

public PCLDataType lfracts( PCLDataType b ) {

    if ( b instanceof PCLInt )
        return new PCLInt( var / intValue(b) );
    return new PCLDouble( var / PCLDouble.doubleValue(b) );
}

public PCLDataType rfracts( PCLDataType b ) {
    return lfracts( b );
}

public PCLDataType div( PCLDataType b) {
    if ( b instanceof PCLInt )
        return new PCLInt( var / intValue(b) );
    return new PCLDouble( var / PCLDouble.doubleValue(b) );
}

public PCLDataType modulus( PCLDataType b ) {
    if ( b instanceof PCLInt )
        return new PCLInt( var % intValue(b) );
    return new PCLDouble( var % PCLDouble.doubleValue(b) );
}


public PCLDataType rem( PCLDataType b ) {
    var %= intValue( b );
    return this;
}

public PCLDataType gt( PCLDataType b ) {
    if ( b instanceof PCLInt )
        return new PCLBool( var > intValue(b) );
    return b.lt( this );
}

public PCLDataType ge( PCLDataType b ) {
    if ( b instanceof PCLInt )
        return new PCLBool( var >= intValue(b) );
    return b.le( this );
}

public PCLDataType lt( PCLDataType b ) {
    if ( b instanceof PCLInt )
        return new PCLBool( var < intValue(b) );
```

```
        return b.gt( this );
    }

    public PCLDataType le( PCLDataType b ) {
        if ( b instanceof PCLInt )
            return new PCLBool( var <= intValue(b) );
        return b.ge( this );
    }

    public PCLDataType eq( PCLDataType b ) {
        if ( b instanceof PCLInt )
            return new PCLBool( var == intValue(b) );
        return b.eq( this );
    }

    public PCLDataType ne( PCLDataType b ) {
        if ( b instanceof PCLInt )
            return new PCLBool( var != intValue(b) );
        return b.ne( this );
    }

    public boolean equals(Object arg0)
    {
       return compareTo(arg0) == 0;
    }

       public int compareTo(Object arg0) {
             return new Integer(this.var).compareTo(new
Integer(((PCLInt)arg0).var));
       }

       public PCLDataType isTypeOf(PCLString regex)
       {
        Pattern pattern = Pattern.compile(regex.var);
        Matcher matcher = pattern.matcher(""+this.var);

        return new PCLBool(matcher.find());
       }
}
```

PCLInterpreter.java

```
import java.util.*;
import java.io.*;
import java.lang.reflect.Field;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

import antlr.CommonAST;
import antlr.collections.AST;
import antlr.RecognitionException;
import antlr.TokenStreamException;
import antlr.TokenStreamIOException;

/** Interpreter routines that is called directly from the tree walker.
 *  Can forward actions and requests to the Charting control
 *
 *  @author T.Chou
 */
class PCLInterpreter {

       // The current symbol table
    HashMap<String, PCLDataType> symt;

    // The global symbol table
    HashMap<String, PCLDataType> g_symt;

    final static int fc_none = 0;
    final static int fc_break = 1;
    final static int fc_continue = 2;
    final static int fc_return = 3;

    private String label;

    private int control = fc_none;

    public PCLDataType currentObj = null;

    public PCLInterpreter() {
        g_symt = new HashMap<String, PCLDataType>();
        g_symt.put("chart", PCL.chart);
        symt = g_symt;
    }

    // Get variable value from the symbol table
    // Or get the member variable value from an object
    public PCLDataType getVariable( String s )
    {
        PCLDataType x = symt.get(s);

       if(currentObj != null)
       {
              Class c = currentObj.getClass();
              Method[] m = c.getMethods();
              for(int i=0; i<m.length; i++)
              {
                     if(m[i].getName().compareTo(s)==0)
                     {
                            PCLFunction func = new PCLFunction(s, 0, currentObj);
                            currentObj = null;
                            return func;
                     }
```

```
            }
            PCLDataType result = null;
            try
            {
                  result = PCLInterpreter.getJavaVariable(currentObj, s);
            }
            finally
            {
                  currentObj = null;
            }
            return result;
      }

       if ( null == x )
            throw new PCLException("Variable " + s + " is not defined");
       return x;
    }

    // Declare a variable
    public PCLDataType setVariable(String s, PCLDataType d)
    {
       if(symt.get(s) == null)
       {
            PCLDataType copy = d.copy();
            copy.name = s;
            symt.put(s, copy);
            return copy;
       }
       else
            throw new PCLException("Variable " + s + " is already defined" );
    }

    // Copy the contents of 'a' and return it
    public PCLDataType rvalue( PCLDataType a )
    {
        if ( null == a.name )
            return a;
        return a.copy();
    }

    // a = b
    public PCLDataType assign( PCLDataType a, PCLDataType b )
    {
        if ( null == a.name )
            throw new PCLException( "cannot assign value to read only object"
);

       if(symt.get(a.name) != null)
       {
            PCLDataType x = rvalue( b );
            x.setName( a.name );
            symt.put( x.name, x);
            return x;
       }
       else
            throw new PCLException( "Variable " + a.name + " has not been
defined" );
    }

    // User reflection to get a Java variable
    public static PCLDataType getJavaVariable(PCLDataType obj, String varName)
    {
       Class c = obj.getClass();
```

```
      if(varName.startsWith("_"))
            throw new PCLException("Access denied. You do not have permission
to access this type of variable on this object: "+obj);

      try {
                  Field f = c.getField(varName);
                  Object result;
                  try {
                        result = f.get(obj);
                        if(result instanceof PCLDataType)
                              return (PCLDataType)result;
                        else
                              throw new PCLException("Invalid Field Type");
                  } catch (IllegalArgumentException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                  } catch (IllegalAccessException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                  }
            } catch (SecurityException e) {
                  // TODO Auto-generated catch block
                  e.printStackTrace();
            } catch (NoSuchFieldException e) {
                  // TODO Auto-generated catch block
                  e.printStackTrace();
            }

            throw new PCLException("No such member variable");
    }

    // Invoke a Validator or Generator function
    public PCLDataType invokePCLFunc(PCLFunction func, PCLDataType params[])
throws antlr.RecognitionException
    {
       try
       {
              // heck numbers of actual and formal arguments
              String[] args = ((PCLFunction)func).getArgs();
              if ( args.length != params.length )
                  return func.error( "unmatched length of parameters" );

              // set current symbol table to be function symbol table
              symt = func.getSymbolTable();

              // copy args to symbol table
              for ( int i=0; i<args.length; i++ )
              {
                  PCLDataType d = rvalue( params[i] );
                  d.setName( args[i] );
                  symt.put( args[i], d);
              }

              // call the function body
              PCLDataType r = PCL.walker.expr( ((PCLFunction)func).getBody() );

              // no break or continue can go through the function
              if ( control == fc_break || control == fc_continue )
                  throw new PCLException( "nowhere to break or continue" );

              // if a return was called
              if ( control == fc_return )
```

```
                tryResetFlowControl( ((PCLFunction)func).name );

             return r;
      }
      finally
      {
            // Reset symbol table to global symbol table
            symt = g_symt;
      }
   }

   // User reflection to call a function on a object
   public static PCLDataType invokeJavaFunc(PCLDataType obj, String funcName,
Object[] params)
   {
      Class c = obj.getClass();

      Class p[] = new Class[params.length];
      String debug = funcName + "(";
      for(int i=0;i<params.length; i++)
      {
            p[i] = params[i].getClass();
            debug += p[i].toString();
            if(i != params.length - 1)
                  debug += ", ";
      }
      debug += ")";
      Method m = null;
      try {
                  m = c.getMethod(funcName, p);
            } catch (SecurityException e) {
                  // TODO Auto-generated catch block
                  e.printStackTrace();
                  return null;
            } catch (NoSuchMethodException e) {
                  throw new PCLException("Function call " + debug + " is not
defined for object "+obj.name + " or has invalid arguments");
            }

            // Only functions that return PCLDataTypes are allowed
//          if(!m.getReturnType().getSuperclass().equals(PCLDataType.class))
            if(!PCLDataType.class.isAssignableFrom(m.getReturnType()))
                  throw new PCLException("Internal Function. Cannot call
functions that do not have a PCLDataType as a return type: "+obj.name);

            try {
                  return (PCLDataType)m.invoke(obj, params);
            } catch (IllegalArgumentException e) {
                  // TODO Auto-generated catch block
                  e.printStackTrace();
            } catch (IllegalAccessException e) {
                  // TODO Auto-generated catch block
                  e.printStackTrace();
            } catch (InvocationTargetException e) {
                  // TODO Auto-generated catch block
                  e.printStackTrace();
            }

            throw new PCLException("Unable to call function: "+funcName);
   }

   // Used to access collection items using the [] operator
   public PCLDataType accessCollection(PCLDataType obj, PCLDataType key)
```

```
    {
       if(obj instanceof PCLCollection)
       {
              return ((PCLCollection)obj).get(key);
       }
       throw new PCLException("object is not a collection");
    }

    public PCLDataType funcInvoke(PCLAntlrWalker walker,  PCLDataType func,
PCLDataType[] params ) throws antlr.RecognitionException
    {
       // func must be an existing function
        if ( !( func instanceof PCLFunction ) )
            return func.error( "not a function" );

        PCLFunction f = (PCLFunction)func;
       if(f.owner != null)
       {
              PCLDataType result;
              try
              {
                     // Generate functions are special because they have fixed
args followed by a variable number args
                     // Won't be able to use reflection here to call function
because Java can't handle
                     // a combination of fixed and variable args very well
                     if(f.name.compareTo("generate")==0 && f.owner instanceof
PCLChart)
                     {
                            // First 3 should be generator name, series name, and
plot type
                            if(params.length < 3)
                                   throw new PCLException("generate function
accepts at least 3 arguments");

                            //System.out.println(params[0]);
                            if(!(params[0] instanceof PCLGenerator))
                                   throw new PCLException("Not a valid
generator");

                                   if(!(params[1] instanceof PCLString))
                                   throw new PCLException("Not a valid name for
data series");

                                   if(!(params[1] instanceof PCLString))
                                   throw new PCLException("Not a valid name plot
type");

                            PCLDataType[] optionalParams = new
PCLDataType[params.length - 3];
                            for(int i=3; i<params.length; i++)
                                   optionalParams[i-3] = params[i];

                            result = PCL.chart.generate(
                                          (PCLGenerator)params[0],
                                          (PCLString)params[1],
                                          (PCLString)params[2],
                                          optionalParams
                                          );
                     }
                     else
                            result = invokeJavaFunc(f.owner, func.name, params);
              }
```

```
            finally
            {
            }
            return result;
      }
      else
      {
           return invokePCLFunc(f, params);
      }
   }

   // Register a validator with the symbol table
   public void validatorRegister( String name, AST body )
   {
      if(symt.get(name) != null)
            throw new PCLException(name + " is already used for another
object");

      symt.put( name, new PCLValidator( name, body ) );
   }

   // Register a generator with the symbol table
   public void generatorRegister( String name, String[] args, AST body )
   {
      if(symt.get(name) != null)
            throw new PCLException(name + " is already used for another
object");
       symt.put( name, new PCLGenerator( name, args, body ) );
   }

   public void tryResetFlowControl( String loop_label )
   {
       if ( null == label || label.equals( loop_label ) )
           control = fc_none;
   }

   public void loopNext( String loop_label )
   {
       if ( control == fc_continue )
           tryResetFlowControl( loop_label );
   }

   public void loopEnd( String loop_label )
   {
       if ( control == fc_break )
           tryResetFlowControl( loop_label );
   }

   public void setBreak( String label )
   {
       this.label = label;
       control = fc_break;
   }

   public void setContinue( String label )
   {
       this.label = label;
       control = fc_continue;
   }

   public void setReturn( String label )
   {
       this.label = label;
```

```
        control = fc_return;
    }

    public boolean canProceed()
    {
        return control == fc_none;
    }

    public PCLDataType[] convertExprList( Vector v )
    {
        /* Note: expr list can be empty */
        PCLDataType[] x = new PCLDataType[v.size()];
        for ( int i=0; i<x.length; i++ )
            x[i] = (PCLDataType) v.elementAt( i );
        return x;
    }

    public static String[] convertVarList( Vector v )
    {
        /* Note: var list can be empty */
        String[] sv = new String[ v.size() ];
        for ( int i=0; i<sv.length; i++ )
            sv[i] = (String) v.elementAt( i );
        return sv;
    }

    public static PCLDataType getNumber( String s )
    {
        if ( s.indexOf( '.' ) >= 0
             || s.indexOf( 'e' ) >= 0 || s.indexOf( 'E' ) >= 0 )
            return new PCLDouble( Double.parseDouble( s ) );
        return new PCLInt( Integer.parseInt( s ) );
    }
}
```

PCLString.java

```
import java.io.PrintWriter;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

/**
 * the wrapper class for string
 * @author Hanhua
 *
 * modified by T.Chou
 */
class PCLString extends PCLDataType
       implements Comparable
{
    String var;

    public PCLString( String str ) {
        this.var = str;
    }

    public String typename() {
        return "string";
    }

    public PCLDataType copy() {
        return new PCLString( var );
    }

    public void print( PrintWriter w ) {
        if ( name != null )
            w.print( name + " = " );
        w.print( var );
        w.println();
    }

    public PCLDataType plus( PCLDataType b ) {
        if ( b instanceof PCLString )
            return new PCLString( var + ((PCLString)b).var );

        return error( b, "+" );
    }

    public PCLDataType add( PCLDataType b ) {
        if ( b instanceof PCLString )
        {
            var = var + ((PCLString)b).var;
            return this;
        }

        return error( b, "+=" );
    }

    public boolean equals(Object arg0)
    {
       return compareTo(arg0) == 0;
    }

       public int compareTo(Object arg0) {
             return this.var.compareTo(((PCLString)arg0).var);
       }
```

```
       public PCLDataType isTypeOf(PCLString regex)
       {
        Pattern pattern = Pattern.compile(regex.var);
        Matcher matcher = pattern.matcher(this.var);

        return new PCLBool(matcher.find());
       }
}
```

PCLUniqueID.java

```
/*
 * @author C.Choong
 */
public class PCLUniqueID {
    static long current = System.currentTimeMillis();

    static public synchronized long get() {
        return current++;
    }
}
```

PCLValidator.java

```
import antlr.collections.AST;

/*
 * @author T.Chou
 */
public class PCLValidator extends PCLFunction
{
    public PCLValidator( String name, AST body)
    {
       super(name, new String[0], body);
    }

    // Places the value in the symbol table so
    // that the Interpreter can use it
    public void setValue(PCLDataType v)
    {
       v.name = "value";
       pst.put("value", v);
    }

    // Gets the value from the Symbol table
    public PCLDouble getValue()
    {
       PCLDataType v = pst.get("value");
       if(v instanceof PCLInt)
              return new PCLDouble(((PCLInt)v).var);
       else if(v instanceof PCLDouble)
              return new PCLDouble(((PCLDouble)v).var);
       else
              throw new PCLException("Cannot assign <value> object to a non-
numeric data type");
    }

}
```

PCLVariable.java

```
import java.io.PrintWriter;

/**
 * The wrapper class for unsigned variables
 *
 * @author Hanhua
 * modified by T. Chou
 */
class PCLVariable extends PCLDataType {
    public PCLVariable( String name ) {
        super( name );
    }

    public String typename() {
        return "undefined-variable";
    }

    public PCLDataType copy() {
        throw new PCLException( "Variable " + name + " has not been defined" );
    }

    public void print( PrintWriter w ) {
        w.println( name + " = <undefined>" );
    }
}
```