PCL

Portable Charting Language

Language Reference Manual

Chee Seng Choong Thomas Chou

1.1 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.

1.1.1 Comments

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

1.1.2 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.

1.1.3 Keywords

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

```
if          else        for         value       validate
generate    return      break       continue    load
save        let         print       true        false
in          newseries   scatter     line        bar
```

1.1.4 Constants

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

1.1.4.1 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 "-".

1.1.4.2 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.

1.1.6 Other tokens

The following operators or symbolic characters are used:

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

1.2 Types

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

1.3 Expressions

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

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

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

1.2.4 `( 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.

1.2.5 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.

1.2.6 Unary operators
Expressions with unary operators group left to right.

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

1.2.6.2 !expression
The result is the logic negation of the boolean value of the expression.

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

1.2.7.1 expression * expression
The binary * operator indicates multiplication.

1.2.7.2 expression / expression
The binary / operator indicates division.

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

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

1.2.8.1 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.

1.2.8.2 expression – expression
The result is the difference of the operands.

1.2.9 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.

1.2.10 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.

1.2.11 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.

1.2.12 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.

1.2.13 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.

1.2.14 Declarators
A declaration begins with the keyword `let` followed by a single or list of comma-separated identifiers.

1.2.15 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`.

1.2.15.1 Expression statement
Most statements are expression statements of the form
`expression;`

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

1.2.15.3 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`.

1.2.15.4 For statement

The for statement has the form

`for ( expression1 ; expression2 ; expression3 )` statement

The first expression 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.

1.2.15.5 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.

1.2.15.6 Continue statement

The statement

```
continue ;
```

causes control to pass to the end of the current iteration of the smallest enclosing `for` loop.

1.2.15.7 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.

1.3 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`	`name`	`plot`
	`bgcolor`	`save`

```
            fgcolor           addseries
            datacolumn        removeseries
            dataseries
Dataseries  name              add
            x
            y
            type
DataColumn  name
            data
```

Global functions:

Function name	Parameters	Description
`print`	*string*	Prints the string to console
`load`	*string*	Loads a csv data file and creates a Chart object

1.3.1 Collections

A Collection represents an object 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"]
```

Otherwise can be accessed by an integer index

Example:

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

1.3.2 Chart

The Chart object is a construct used to store the data that is to be graphed. It is created by calling a predefined function `load()` which takes in the filename of the charting data file.

Example:

```
let myChart = 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 properties of the Chart object are described below.

Property Name	Description
`name`	A unique identifier that can be specified by the user.
`bgcolor`	Background color of the chart.
`fgcolor`	Foreground color of the chart.
`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	Descripition
`addseries`	series name	Creates a new Dataseries object to be plotted.
`removeseries`	series name	Remove a DatSeries object from the dataseries collection.
`save`	filename	Saves the Chart object's data into a csv file.
`plot`	series name	Plots the data series specified. If no parameter is passed in, then all data series are plotted.

1.3.3 Dataseries

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

Property Name	Descripition
`name`	A unique identifier that can be specified by the user
`x`	A Collection of values which is mapped to one of the data columns in the Chart object.
`y`	A Collection of values which is mapped to one of the data columns in the Chart object.
`type`	Describes the type of plot this data series will be. The possible types are {Scatter, Line, Bar}

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

1.3.4 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 is automatically assigned a name. The name has the following format `col1, col2, col3`, ...

Property Name	Description
`name`	A unique identifier that can be specified by the user
`data`	An array of values stored

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

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

// Assign chart attributes
chart.bgcolor = #000000
```

```
chart.fgcolor = #FFFFFF
chart.datacolumn["col1"].name = "X values"
chart.datacolumn["col2"].name = "Y values"

// Create a new data series
chart.addseries("new series")
chart.dataseries["new series"].x = chart.datacolumn["X values"].data
chart.dataseries["new series"].y = chart.datacolumn["Y values"].data
chart.dataseries["new series"].type = Scatter

// Save the chart
chart.save("newfile.csv")

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

1.4 Validators and Generators

1.4.1 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
}

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 only terminate on a `return` statement. A return value of true will indicate that validation for current cell was successful and false otherwise. 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:

```
      validate("My Validation Rule", "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.

Example:

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

The special value identifier also has a function called `isTypeOf()` which can be used to check the format of a specific value. The function takes in a regular expression string describing the format that the user wants each data point to validate against.

Example:

```
[FormatCheck]
{
	// Check to see if a value is a number or not
	if(value.isTypeOf("[0...9]")
		return true
	else
		return false
}
```

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
}
```

1.4.2 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 commans and within the parantheses.

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, chart object and list of parameters:

```
generate(generator-name, chart-object, 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
	newseries.name = "Average"
	for(int i=0; i<high.x.length; i++)
	{
		newseries.add(high.x[i], (high.y[i] + low.y[i]) / 2);
	}

	return true;
}
```

1.4.3 Importing Validators and Generators

A user can import Validators and Generators defined in a different file using the import command shown below:

```
import <filename of validators and generators>
```

Example:

```
	import "PCL-Modules.pcl"
```