Javacc grammar for PCD (update Aug. 21, 2012)

A. Javacc grammar for PCD (BioLegato dialect)

```
options {
/*     JAVA_UNICODE_ESCAPE=true;
       UNICODE_INPUT=true;*/
       IGNORE_CASE=true;
       KEEP_LINE_COLUMN=true;
       STATIC=false;
}



PARSER_BEGIN(PCD)

package org.biolegato.menu ;

import java.awt.image.BufferedImage ;
import java.io.File               ;
import java.io.FileReader         ;
import java.io.FileInputStream    ;
import java.io.IOException        ;
import java.io.InputStreamReader  ;
import java.io.BufferedReader     ;
import java.util.Set              ;
import java.util.Map              ;
import java.util.List             ;
import java.util.HashSet          ;
import java.util.HashMap          ;
import java.util.ArrayList        ;
import java.util.LinkedList       ;
import java.util.LinkedHashMap    ;
import javax.swing.Action         ;
import javax.swing.BoxLayout      ;
import javax.swing.JButton        ;
import javax.swing.JFrame         ;
import javax.swing.JMenu          ;
import javax.swing.JList          ;
import javax.swing.JPanel         ;
import javax.swing.JSlider        ;
import javax.swing.JTabbedPane    ;
import javax.swing.JTextField     ;
import javax.swing.JMenuItem      ;
import javax.swing.ImageIcon      ;
import javax.imageio.ImageIO      ;
import org.biolegato.main.BLMain  ;
import org.biolegato.main.PluginLoader  ;
import org.biolegato.main.PluginWrapper ;
import org.biolegato.main.StreamCopier  ;

/**
 * A class used to parse PCD files into BioLegato.
 */
public class PCD {
```

```
	/**
	* The menu item name for the current PCD file program
	*/
	private String  name             = null  ;
	/**
	* If the exec parameter in a PCD file is set, this variable will also be set.
	* This variable is used to run commands which do not have any associated display
	* widgets.  If this is variable is not null, the command will run once the menu
	* button is pressed.  The command used for running will be stored in this
variable.
	*/
	private String  exec             = null  ;
	/**
	* The menu item icon for the current PCD file program
	*/
	private String  icon             = null  ;
	/**
	* The menu item tooltip text for the current PCD file program
	*/
	private String  tooltip          = null  ;
	/**
	* Stores whether the current PCD file is supported on the current operating
	* system.
	*/
	private boolean systemSupported   = true  ;
	/**
	* Stores the current BioLegato canvas object.
	*/
	private BLMain canvas            = null  ;
	/**
	* The widget list to use for running the current PCD command.  This is used
	* for replacing any command line variables.
	*/
	private Map<String, Widget> masterWidgetList;
	/**
	* This array is used for the template toArray method for lists
	* i.e. toArray(BLANK_STRING_ARRAY)
	*/
	private static final String[] BLANK_STRING_ARRAY = new String[0];
	/**
	* The width to display the icons in BioLegato menus
	*/
	private static final int ICONW = 16;
	/**
	* The height to display the icons in BioLegato menus
	*/
	private static final int ICONH = 16;
	/**
	* An enumeration to store which type of list to create using list parameters.
	* To elaborate, this enumeration is utilized because the listParameter method,
	* which reads in the parameters for any PCD list, is reused for comboboxes,
	* choosers and choice lists.
	*/
	public static enum ListType {
	CHOOSER,
```

```
	COMBOBOX,
	LIST;
	}

	/**
	* Loads the entire PCD menu structure into BioLegato
	*/
	public static void loadPCD (BLMain canvas) {
	PCD parser = null;

	for (String path : canvas.toPathList(canvas.getProperty("pcd.menus.path"))) {
		loadPCDPath(canvas, new File(path));
	}
	if (!"".equals(canvas.getProperty("pcd.exec"))) {
		try {
		Process p =
Runtime.getRuntime().exec(canvas.envreplace(canvas.getProperty("pcd.exec")).split("\\s
"));
		p.getOutputStream().close();
		parser = new PCD(new InputStreamReader(p.getInputStream()));
		parser.canvas = canvas;
		parser.parseFullMenu(0, new File(BLMain.envreplace("$BL_HOME")));
		} catch (Throwable th) {
		th.printStackTrace(System.err);
		}
	}
	}

	/**
	* Loads a path of PCD files into BioLegato
	**
	* @param path the path of the PCD file(s) to load
	*/
	private static void loadPCDPath (BLMain canvas, File path) {
	File orderfile;
	String tpath = path.getAbsolutePath().toLowerCase();

	if (path.exists() && path.canRead()) {
		///////////////
		// RECURSION //
		///////////////
		// 1.  if the path parameter is  a directory, loop through all of the
files
		//     and subdirectories stored in path, and run loadPCD path on them.
		// 2.  if the path parameter is a file, read the PCD file.
		if (path.isDirectory()) {
		orderfile = new File(path, "pcd_order");
		if (orderfile.exists() && orderfile.isFile() && orderfile.canRead()) {
			try {
				String line;
				File ofile;
				Set fset = new HashSet();
				BufferedReader oread = new BufferedReader(new
FileReader(orderfile));
```

```
                        while ((line = oread.readLine()) != null) {
                        if (!line.trim().equals("")) {
                              ofile = new File(path, line);
                              if (ofile.exists() && ofile.canRead()) {
                              if (ofile.isDirectory() || line.endsWith("/")) {
                                    canvas.addMenuHeading(line);
                              }
                              loadPCDPath(canvas, ofile);
                              fset.add(ofile);
                              }
                        }
                        }
                        for (File subdir : path.listFiles()) {
                        if (!fset.contains(subdir)) {
                              loadPCDPath(canvas, subdir);
                        }
                        }
                  } catch (IOException ioe) {
                        BLMain.error("Error reading pcd_order, skipping pcd_order
preferences!", "PCD Parser");
                        ioe.printStackTrace(System.err);
                        for (File subdir : path.listFiles()) {
                        loadPCDPath(canvas, subdir);
                        }
                  }
            } else {
                  for (File subdir : path.listFiles()) {
                        loadPCDPath(canvas, subdir);
                  }
            }
            } else if (tpath.endsWith(".pcd") || tpath.endsWith(".blitem") ||
tpath.endsWith(".blmenu")) {
            try {
                  PCD parser = new PCD(new FileInputStream(path));
                  parser.canvas = canvas;
                  JMenuItem jmiresult = parser.parseMenuItem(0,
path.getParentFile());
                  if (jmiresult != null) {

      canvas.addMenuHeading(path.getParentFile().getName()).add(jmiresult);
                  }
                  if (canvas.debug) {
                        System.out.println("PARSE OK! - " + path);
                  }
            } catch (Throwable th) {
                  System.err.println("PARSE FAILED! - " + path);
                  th.printStackTrace(System.err);
                  System.err.flush();
                  System.err.flush();
            }
            } else if (tpath.endsWith(".class")) {
            try {
                  /*
                  * Handles reading in class files instead of PCD
```

```
                    * (this feature can be enabled from the properties file for
BioLegato).
                    */
                    Class [] jframeClass  = new Class [] { BLMain.class         };
                    Object[] jframeObject = new Object[] { canvas };
                    Map<String, PluginWrapper> pluginHash = new HashMap<String,
PluginWrapper>();
                    PluginLoader.loadClasses(pluginHash,
path.getParentFile().toURI().toURL(), path.getName().substring(0,
path.getName().length() - 6));

                    // Load the class menus
                    for (PluginWrapper plugin : pluginHash.values()) {
                           if (!plugin.getName().contains("$")) {
                           if (plugin.isA(JMenuItem.class)) {
                                  try {

      canvas.addMenuHeading(path.getParentFile().getName()).add((JMenuItem)
plugin.create(jframeClass, jframeObject));
                                  } catch (Throwable th) {
                                  BLMain.error("error loading the plugin menu: " +
plugin.getName(), "BLMain");
                                  th.printStackTrace(System.err);
                                  }
                           } else if (plugin.isA(Action.class)) {
                                  try {

      canvas.addMenuHeading(path.getParentFile().getName()).add(new
JMenuItem((Action) plugin.create(jframeClass, jframeObject)));
                                  } catch (Throwable th) {
                                  BLMain.error("error loading the plugin menu: " +
plugin.getName(), "BLMain");
                                  th.printStackTrace(System.err);
                                  }
                           }
                           }
                    }
             } catch (Throwable th) {
                    BLMain.error("error loading the class: " + path, "BLMain");
                    th.printStackTrace(System.err);
             }
             } else if (tpath.endsWith(".jar")) {
             try {
                    /*
                    * Handles reading in class files instead of PCD
                    * (this feature can be enabled from the properties file for
BioLegato).
                    */
                    Map<String, PluginWrapper> pluginHash = new HashMap<String,
PluginWrapper>();
                    PluginLoader.loadJar(pluginHash, path);
                    // Load the class menus
                    for (PluginWrapper plugin : pluginHash.values()) {
                           if (plugin.isA(JMenu.class)) {
                           try {
```

```
                                canvas.getJMenuBar().add((JMenu) plugin.create());
                          } catch (Throwable th) {
                                BLMain.error("error loading the plugin: " +
plugin.getName(), "BLMain");
                                th.printStackTrace(System.err);
                          }
                          }
                    }
             } catch (Throwable th) {
                    BLMain.error("error loading the jar file: " + path, "BLMain");
                    th.printStackTrace(System.err);
             }
             }
       } else if (path.isDirectory() || tpath.endsWith(".pcd") ||
tpath.endsWith(".blitem")
             || tpath.endsWith(".class") || tpath.endsWith(".jar")) {
             System.out.println("ERROR - cannot read \"" + path.getAbsolutePath() +
"\"");
       }
       }
}

PARSER_END(PCD)



/* PRODUCTIONS */


/**
 * Parses a PCD menu.
 **
 * The format for a PCD menu is:
 * menu Name
 * MenuItem data
 */
void parseFullMenu(int scope, File home) :
{
       /* The name of the menu */
       String menuName;
       /* The current menu item being parsed */
       JMenuItem jmiresult;
}
{
       ( LOOKAHEAD({testIndent(scope)})
       ( <T_MENU> <WSP> menuName=Text() nl()
             ( LOOKAHEAD({testIndent(scope + 1)})
             (
                    <T_ITEM> nl()
                    {
                          try {
                          jmiresult=parseMenuItem(scope + 2, home);
                          if (jmiresult != null) {
                                canvas.addMenuHeading(menuName).add(jmiresult);
                          }
```

```
                        } catch (ParseException ex) {
                        System.out.println("FAILED PARSE OF MENU ITEM --- SKIPPING
AHEAD!");
                        ex.printStackTrace(System.err);
                        Token skipto;
                        do {
                               skipto = getNextToken();
                        } while (skipto.kind != T_ITEM && skipto.kind != T_MENU
                               && skipto.kind != EOF);
                        }
                  }
            )
            )+
      )
      )+
}

/**
 * Parses a PCD menu item.
 **
 * The format for a PCD menu item is:
 * Header
 * Content
 */
JMenuItem parseMenuItem(int scope, File home) :
{
      /* The resulting menu item that was parsed from the file */
      JMenuItem jmiresult = null;
      /* A list of widgets parsed for the menu item's creation */
      Map<String,Widget> widgetList;
      /* A file object to ensure a proper icon file path exists */
      File imageFile = null;
      /* The icon file name for displaying on the menu */
      ImageIcon imageIcon = null;
      /* The icon data for displaying in the parameters window for the menu item*/
      BufferedImage image = null;
}
{
      {
      name=null;
      exec=null;
      icon=null;
      tooltip=null;
      systemSupported=true;
      }
      Header(scope)
      widgetList=Body(scope)
      {
      if (systemSupported) {
            try {
            if (icon != null && !"".equals(icon)) {
                  if (icon.startsWith("/") || icon.startsWith("$")) {
                        imageFile = new File(BLMain.envreplace(icon));
                  } else {
                        imageFile = new File(home, BLMain.envreplace(icon));
```

```
                    }

                    if (imageFile != null && imageFile.exists() && imageFile.canRead()
&& imageFile.isFile()) {
                            BufferedImage fileImage = ImageIO.read(imageFile);
                            // solution adapted from:
http://stackoverflow.com/questions/6916693/jmenuitem-imageicon-too-big
                            if (fileImage != null) {
                            image = new BufferedImage(ICONW, ICONH,
BufferedImage.TYPE_INT_RGB);

       image.getGraphics().drawImage(fileImage,0,0,ICONW,ICONH,null);
                            imageIcon = new ImageIcon(image);
                            }
                    }
             }
             } catch (Exception e) {
             BLMain.error("Invalid image format: " + icon, "PCD parser");
             imageIcon = null;
             }

             if (imageIcon != null) {
             jmiresult = new JMenuItem(name, imageIcon);
             } else {
             jmiresult = new JMenuItem(name);
             }
             if (tooltip != null && !"".equals(tooltip)) {
             jmiresult.setToolTipText(tooltip);
             }

             /* Determine whether to create a new window on clicking the menu item
             * or directly run the command specified in the PCD file's exec parameter
             * ---
             * this is determined by wheter the exec parameter is specified in the PCD
file! */
             if (exec != null) {
             jmiresult.addActionListener(new CommandThread(exec, widgetList, canvas));
             } else {
             jmiresult.addActionListener(new RunWindow(name, widgetList, canvas,
image));
             }
       } else {
             System.out.println("System not supported");
       }
       return jmiresult;
       }
}

/**
 * Parses a PCD menu item header (good for reading just the basic data of a PCD file).
 **
 * The format for a PCD menu item header is:
 * [ Optional Blank Space ]
 * [ PCD Options ]
 * Tabs and Parameters
```

```
 **
 * Currently supported PCD options:
 *
 * <table>
 * <tr><th>Option name</th> <th>Description</th></tr>
 * <tr><td>name</td>       <td>the name of the PCD command</td></tr>
 * <tr><td>tip</td>        <td>the tool-tip text for the PCD command</td></tr>
 * <tr><td>icon</td>       <td>the path of the PCD command's icon file</td></tr>
 * <tr><td>system</td>     <td>a list of supported system configurations
 *                              for the PCD command</td></tr>
 * </table>
 */
void Header(int scope) :
{
       /* the token to store all of the information received about the option */
       Token t;
}
{
       /* Match any preceding whitespace (note that nl() tokens skip blank lines */
       [ nl() ]

       /* Match any PCD options - this should come before the actual progarm definition
       * this rule makes things more organized and easier to read */
       LOOKAHEAD({testIndent(scope)})
       <T_CMDNAME> <WSP> { name   = Text(); } nl()

       [ LOOKAHEAD({testIndent(scope) && getToken(1).kind == T_ICON})
       <T_ICON>     <WSP> { icon  = Text(); } nl() ]
       [ LOOKAHEAD({testIndent(scope) && getToken(1).kind == T_TIP})
       <T_TIP>      <WSP> { tooltip = Text(); } nl() ]
       [ LOOKAHEAD({testIndent(scope) && getToken(1).kind == T_SYSTEM})
       <T_SYSTEM>
              ( <WSP>                                SystemName() nl()
              | nl() ( { assertIndent(scope + 1); } SystemName() nl() )+
              ) ]
       [ LOOKAHEAD({testIndent(scope) && getToken(1).kind == T_EXEC})
       <T_EXEC>     <WSP> { exec  = Text(); } nl() ]
}


/**
 * Parses PCD menu item content
 **
 * The format for a PCD menu item is:
 * [ Optional Blank Space ]
 * [ PCD Options ]
 * Tabs and Parameters
 */
Map<String, Widget> Body(int scope) :
{
}
{
       /* Initialize the widget list to store all of teh widgets for the program */
       { masterWidgetList = new LinkedHashMap<String, Widget>(); }
```

```
      /* Match any parameters or tabs in the prorgam (i.e. the functional components
of a PCD file) */
      ( Content(scope, masterWidgetList) )*

      /* Match the end of file token */
      [ <EOF> ]

      /* return the master widget list */
      { return masterWidgetList; }
}


/**
 * Parses PCD file content
 **
 * The format for a PCD file is:
 * [ Optional Blank Space ]
 * [ PCD Options ]
 * Tabs and Parameters
 */
void Content(int scope, Map<String, Widget> widgetList) :
{
}
{
      /* Match any parameters or tabs in the prorgam (i.e. the functional components
of a PCD file) */
      (
      Param(scope, widgetList)
      | Tab  (scope, widgetList)
      | Panel(scope, widgetList)
      )
}


/**
 * Generates a tabbed pane based on reading the tab tag from the PCD file.
 **
 * This function reads the <T_TAB> tag, parses the name, and creates a
 * new panel object that all sub-components can be added to.  The tab
 * is then added to a tabbed pane in the main window.
 *
 * Each tab can only contain paramter objects,
 * and each tab MUST contain at least one parameter object.
 **
 * @param widgetList the list of widgets to add the tab to
 */
void Tab(int scope, Map<String, Widget> widgetList) : {
      /* Temporarily stores parameters before they are added to the main panel */
      Map<String, Widget> tabParameterList = new LinkedHashMap<String, Widget>();
      /* The current tabset to add to biolegato's menu system*/
      TabbedWidget tabset = null  ;
}
{
      { assertIndent(scope); } <T_TABSET> nl()
      /* Ensure that the main tab is not null */
```

```
	{
	tabset = new TabbedWidget();
	widgetList.put("____tab" + widgetList.size(), tabset);
	}

	(
	/* Match the tab name and create the tab */
	LOOKAHEAD({testIndent(scope + 1)})
	( <T_TAB> <WSP> { tabset.addTab(Text(), tabParameterList); } nl()

		/* Match one or more contentwidgets for the tab */
		( LOOKAHEAD({testIndent(scope + 2)}) Content(scope + 2, tabParameterList)
)+
	) {
		tabParameterList = new LinkedHashMap<String, Widget>();
	}
	)+
}


/**
 * Generates a tabbed pane based on reading the tab tag from the PCD file.
 **
 * This function reads the <T_PANEL> tag, parses it, and creates a panel
 **
 * Panels are used so related parameters can be positioned together
 * for example, related buttons can be positioned side by side
 **
 * @param widgetList the list of widgets to add the tab to
 */
void Panel(int scope, Map<String, Widget> widgetList) : {
	/* The panel widget list to add parameters to */
	Map<String, Widget> panelWidgetList = new LinkedHashMap<String, Widget>();
}
{
	{ assertIndent(scope); } <T_PANEL> nl()
	(
	/* Match one or more contentwidgets for the panel */
	( LOOKAHEAD({testIndent(scope + 1)}) Content(scope + 1, panelWidgetList) )+
	)+
	{ widgetList.put("___panel" + widgetList.size(), new
PanelWidget(panelWidgetList)); }
}


/**
 * Generates a parameter component according to the PCD file's
 * <T_PARAM> production(s).
 **
 * This function reads the <T_PARAM> tag, parses the name, and creates a
 * new parameter component corresponding to the type of parameter read.
 *
 * Each parameter MUST contain a type as its first field!
 *
 * Currently the following types are supported:
```

```
 * <table>
 *  <tr> <th>Type field</th> <th>Description</th> </tr>
 *  <tr> <td>button</td>   <td>Buttons which can run commands or perform
                                functions</td> </tr>
 *  <tr> <td>list</td>     <td>A JList containing options</td> </tr>
 *  <tr> <td>chooser</td>  <td>A radio button field</td> </tr>
 *  <tr> <td>text</td>     <td>A text-field</td> </tr>
 *  <tr> <td>number</td>   <td>A slider/spinner combination to set numbers</td> </tr>
 *  <tr> <td>decimal</td>  <td>A decimal number widget</td> </tr>
 *  <tr> <td>file</td>     <td>A file used for I/O</td> </tr>
 *  <tr> <td>dir</td>      <td>A directory used for file I/O</td> </tr>
 * </table>
 **
 * @param scope the scope to parse the parameter object in
 * @param widgetList the list of widgets to add the parameter to
 */
void Param(int scope, Map<String, Widget> widgetList) :
{
       /* The name of the parameter (for variable reference) */
       String name;

       /* Temporarily stores parameters before they are returned */
       Widget parameter = null;
}
{
       /* Match the tab name and header */
       assertIndent(scope)       <T_PARAM> <WSP> name=Text() nl()

       /* Match one or more option fields for the parameter (NOTE: the type field is
mandatory!) */
       assertIndent(scope + 1) <T_TYPE>  <WSP>
                     (
                            <T_BUTTON>   nl() parameter = buttonFields   (scope + 1)
                     | <T_CHOOSER>  nl() parameter = listFields     (scope + 1,
ListType.CHOOSER)
                     | <T_COMBOBOX> nl() parameter = listFields     (scope + 1,
ListType.COMBOBOX)
                     | <T_LIST>   nl() parameter = listFields      (scope + 1,
ListType.LIST)
                     | <T_TEXT>   nl() parameter = textFields      (scope + 1)
                     | <T_NUMBER>   nl() parameter = numberFields   (scope + 1)
                     | <T_DECIMAL>  nl() parameter = decimalFields  (scope + 1)
                     | <T_FILE>   nl() parameter = fileFields      (scope + 1)
                     | <T_DIR>    nl() parameter = dirFields       (scope + 1)
                     | <T_TEMPFILE> nl() parameter = tempfileFields (scope + 1)
                     )
//     [ <T_CHECK>  ConditionList(scope + 1) ]
       { widgetList.put(name, parameter); }
}


/**
 * Parses all of the fields that should be part of any button field
 **
 * @param scope the scope level to read the objects at
```

```
 */
Widget buttonFields(int scope) :
{
       /* the label for the field */
       String label = "";
       /* the shell command to run */
       String shell = "";
       /* whether the button should close the command window */
       boolean close = false;
}
{
       assertIndent(scope)
       [ <T_LABEL>  <WSP> label=Text() nl() assertIndent(scope)]
       <T_SHELL>    <WSP> shell=Text() nl()
       [ LOOKAHEAD( { testIndent(scope) } )
       <T_CLOSE>    <WSP> close=Bool() nl() ]

       { return new CommandButton(masterWidgetList, label, shell, close, canvas); }
}


/**
 * Parses all of the fields that should be part of any list object
 **
 * @param scope the scope level to read the objects at
 */
Widget listFields(int scope, ListType lType) :
{
       /* the label for the field */
       String label = "";
       /* The default for the text field */
       int value = 0;
       /* The name  of the current choice to add to the choices hashtable */
       String choiceName;
       /* The value of the current choice to add to the choices hashtable */
       String choiceValue;
       /**
       * Used for storing variable choice names
       */
       List<String> choicenames = new LinkedList<String>();
       /**
       * Used for storing variable choice values
       */
       List<String> choicevalues = new LinkedList<String>();
       /**
       * The list widget parsed by the function call
       */
       ListWidget result = null;
}
{
       { assertIndent(scope); }
       [ <T_LABEL>  <WSP> { label = Text();   } nl() { assertIndent(scope); } ]
       [ <T_DEFAULT>   <WSP> { value = Number(); } nl() { assertIndent(scope); } ]
       <T_CHOICES>   nl()
              ( LOOKAHEAD( { testIndent(scope + 1) } )
```

```
              ( (choiceName=Text() <WSP>
              choiceValue=Text() nl() )
                     { choicenames.add(choiceName);
                     choicevalues.add(choiceValue);
              })
              )+
       {
       String[] choicevaluearray = choicevalues.toArray(BLANK_STRING_ARRAY);
       String[] choicenamearray  = choicenames.toArray(BLANK_STRING_ARRAY);

       if (lType == ListType.CHOOSER) {
              result = new Chooser(label, choicenamearray, choicevaluearray, value);
       } else if (lType == ListType.LIST) {
              result = new ChoiceList(label, choicenamearray, choicevaluearray, value);
       } else {
              result = new ComboBoxWidget(label, choicenamearray, choicevaluearray,
value);
       }

       return result;
       }
}


/**
 * Parses all of the fields that should be part of any text field
 **
 * @param scope the scope level to read the objects at
 */
Widget textFields(int scope) :
{
       /* the label for the field */
       String label = "";
       /* The default for the text field */
       String value = "";
}
{
       [ LOOKAHEAD(<T_LABEL>)   { assertIndent(scope); } <T_LABEL>   <WSP>
label=Text() nl() ]
       [ LOOKAHEAD(<T_DEFAULT>) { assertIndent(scope); } <T_DEFAULT> <WSP>
value=Text() nl() ]

       /* Return the corresponding JTextField */
       { return new TextWidget(label, value); }
}


/**
 * Parses all of the fields that should be part of any number field
 **
 * @param scope the scope level to read the objects at
 */
Widget numberFields(int scope) :
{
       /* the label for the field */
```

```
        String label = "";
        /* The minimum number allowed */
        int min      = 0;
        /* The maximum number allowed */
        int max      = 500000;
        /* The default for the number field */
        int value    = 0;
}
{
        assertIndent(scope)
        <T_LABEL>    <WSP> label=Text()  nl() assertIndent(scope)
        <T_MIN>      <WSP> min=Number()  nl() assertIndent(scope)
        <T_MAX>      <WSP> max=Number()  nl()
        [ LOOKAHEAD(<T_DEFAULT>) { assertIndent(scope); }
        <T_DEFAULT>   <WSP> value=Number()  nl() ]
        { if (value < min) { value = min; } }
        { if (value > max) { value = max; } }
        /* Return the corresponding JSlider */
        { return new NumberWidget(label, min, max, value); }
}


/**
 * Parses all of the fields that should be part of any decimal field
 **
 * @param scope the scope level to read the objects at
 */
Widget decimalFields(int scope) :
{
        /* the label for the field */
        String label = "";
        /* The minimum number allowed */
        double min;
        /* The maximum number allowed */
        double max;
        /* The default for the number field */
        double value;
}
{
        assertIndent(scope)
        [ <T_LABEL>  <WSP> label=Text()  nl() assertIndent(scope) ]
        <T_MIN>      <WSP> min=Decimal()   nl() assertIndent(scope)
        <T_MAX>      <WSP> max=Decimal()   nl()
        [ LOOKAHEAD(<T_DEFAULT>) { assertIndent(scope); }
        <T_DEFAULT>   <WSP> value=Decimal() nl() ]

        /* Return the corresponding JTextField */
        { return new TextWidget(label, ""); }
}



/**
 * Parses all of the fields that should be part of any file chooser
 **
```

```
 * @param scope the scope level to read the objects at
 */
Widget fileFields(int scope) :
{
       /* the label for the field */
       String label = "";
       /* The default for the text field */
       String value = "";
}
{
                           { assertIndent(scope); } <T_LABEL>   <WSP> label=Text() nl()
       [ LOOKAHEAD(<T_DEFAULT>) { assertIndent(scope); } <T_DEFAULT> <WSP>
value=Text() nl() ]

       /* Return the corresponding JTextField */
       { return new FileChooser(label, value, canvas); }
}


/**
 * Parses all of the fields that should be part of any directory chooser
 **
 * @param scope the scope level to read the objects at
 */
Widget dirFields(int scope) :
{
       /* the label for the field */
       String label = "";
       /* The default for the text field */
       String value = "";
}
{
                           { assertIndent(scope); } <T_LABEL>   <WSP> label=Text() nl()
       [ LOOKAHEAD(<T_DEFAULT>) { assertIndent(scope); } <T_DEFAULT> <WSP>
value=Text() nl() ]

       /* Return the corresponding JTextField */
       { return new DirectoryChooser(label, value, canvas); }
}


/**
 * Parses all of the fields that should be part of any temporary file field
 **
 * @param scope the scope level to read the objects at
 */
Widget tempfileFields(int scope) :
{
       /* Whether or not to add the contents of the window to the file before execution
       * (whether the temporary file is input for a program). */
       boolean input = false;
       /* Whether or not to add the contents of the file to the window after execution.
       * (whether the temporary file is output for a program). */
       boolean output = false;
       /* Determines whether or not to delete the file after execution. */
```

```
       boolean save = false;
       /* Determines whether or not to overwrite the file if it already exists. */
       boolean overwrite = false;
       /* Stores the file format of the file (used for translation). */
       String format = null;
       /* Stores whether the temporary file uses just the current selection within the
cavnas, or the entire data set stored in biolegato (the equivalent of Select-all) */
       boolean selectall = false;
}
{
       { assertIndent(scope); } <T_DIRECTION> <WSP>
              ( <T_IN> { input = true; } |     <T_OUT> { output = true; } )  nl()
       { assertIndent(scope); } <T_FORMAT>     <WSP>  { format=FileFormat(); }  nl()

       [ LOOKAHEAD(<T_SAVE>)      { assertIndent(scope); } <T_SAVE>       <WSP> {
save=Bool();        }  nl() ]
       [ LOOKAHEAD(<T_OVERWRITE>) { assertIndent(scope); } <T_OVERWRITE> <WSP> {
overwrite=Bool();   }  nl() ]
       [ LOOKAHEAD(<T_CONTENT>)   { assertIndent(scope); } <T_CONTENT>   <WSP>
              ( <T_CANVAS> { selectall = true; } | <T_SELECTION> ) nl() ]

       /* Return the corresponding JTextField */
       { return new TempFile(canvas, input, output, save, overwrite, format,
selectall); }
}


String FileFormat() :
{
       String result = "raw";
}
{
       (
       <T_CSV>       { result = "csv"    ; }
       | <T_TSV>     { result = "tsv"    ; }
       | <T_FASTA>   { result = "fasta"   ; }
       | <T_FLAT>    { result = "flat"   ; }
       | <T_GDE>     { result = "gde"    ; }
       | <T_GENBANK>  { result = "genbank" ; }
       | <T_RAW>     { result = "raw"    ; }
       | <T_MASK>    { result = "mask"   ; }
       | result=Text()
       )
       { return result; }
}



/**
 * Parses a list of supported operating systems in a PCD file
 **
 * The list is then compared with the current operating system
 * to see if it is supported by the PCD command.
 **
 * Currently supported operating systems:
```

```
 * ALL       (the command supports any operating system)
 * Linux
 * OSX
 * Solaris
 * Unix      (the command will only work in UNIX-compatible systems)
 * Windows (the command will only work in Windows-compatible systems)
 */
void SystemName() :
{
      /* Stores the status of whether the current operating system is
      * supported by the software represented in the PCD file */
      boolean osSupported = true;

      /* Stores the status of whether the current machine architecture is
      * supported by the software represented in the PCD file */
      boolean archSupported = true;
}
{
      /* match each operating system token and determine whether or not
      * the operating system matches the current OS */
      ( <T_ALL>    { osSupported = true;                                    }
      | <T_LINUX>   { osSupported = (   BLMain.CURRENT_OS == BLMain.OS.LINUX   ); }
      | <T_OSX>    { osSupported = (   BLMain.CURRENT_OS == BLMain.OS.OSX      ); }
      | <T_SOLARIS> { osSupported = (   BLMain.CURRENT_OS == BLMain.OS.SOLARIS ); }
      | <T_UNIX>   { osSupported = ( ! BLMain.CURRENT_OS.isWindows()           ); }
      | <T_WINDOWS> { osSupported = (   BLMain.CURRENT_OS.isWindows()          ); }
      )

      /* handle the optional architecture list */
      [ <WSP> archSupported = ArchList () ]

      /* add the results of the current operating system support test to the
      * final result of whether the current machine can run the PCD file */
      { systemSupported = systemSupported || ( archSupported && osSupported ); }
}


/**
 * Parses a list of supported system architectures in a PCD file
 **
 * The list is then compared with the current system architecture
 * to see if it is supported by the PCD command.
 **
 * @return whether the current system architecture is supported by the PCD command
 */
boolean ArchList() :
{
      /* Stores the status of the current architecture tested */
      boolean temp = false;

      /* Stores the status of the entire list */
      boolean archSupport = false;
}
{
      /* match each system architecture token and determine whether or not
```

```
       * the system architecture matches the current architecture */
       archSupport=ArchName()

       /* handle additional system architecture names*/
       (
       /* handle list spacer */
       <COMMA>

        /* get the system architecture name */
       temp=ArchName()

    /* test system architecture support */
       { archSupport = archSupport || temp; }
       )*

       /* returns the status of the list test */
       { return archSupport; }
}


/**
 * Matches an architecture name and returns whether it is supported by
 * the current architecture.
 **
 * Currently supported machine architectures:
 * ALL       (the command supports any machine architecture
             - may be useful for shell-scripts)
 * X86       (any x86 compatible machine)
 * AMD64   (any amd64 compatible machine)
 * Sparc   (any amd64 compatible machine)
 **
 * @return whether the architecture is supported
 */
boolean ArchName () : {}
{
       ( <T_ALL>    { return true;                                          }
       | <T_X86>    { return (BLMain.CURRENT_ARCH == BLMain.ARCH.X86 ||
                            BLMain.CURRENT_ARCH == BLMain.ARCH.AMD64); }
       | <T_AMD64>  { return (BLMain.CURRENT_ARCH == BLMain.ARCH.AMD64); }
       | <T_SPARC>  { return (BLMain.CURRENT_ARCH == BLMain.ARCH.SPARC); }
       )
}


/**
 * Parses an identifier token from a PCD file into a Java String
 **
 * @return the coresponding Java String object
 */
String Ident () :
{
       /* The token to parse into a String value */
       Token t = null;
}
{
```

```
	/* Match a text token */
	t=<ID>

	/* Return the token's "image" field */
	{ return t.image; }
}


/**
 * Parses a text token from a PCD file into a Java String
 **
 * @return the coresponding Java String object
 */
String Text () :
{
	/* The token to parse into a String value */
	Token t = null;
}
{
	/* Match a text token */
	t=<TEXT>

	/* Return the token's "image" field */
	{ return t.image.substring(1, t.image.length() - 1).replaceAll("\"\"", "\""); }
}


/**
 * Parses a decimal number from a PCD file into a Java double
 **
 * @return the coresponding Java double value
 */
double Decimal () :
{
	/* The double value parsed by the function */
	double value = 0d;

	/* The token to parse into a double value */
	Token t = null;
}
{
	/* Match a decimal token to parse, then parse
	* the token into a Java integer value
	* - OR -
	* Call the Number() function to parse an integer
	* (Integers are considered decimal numbers, too) */
	( t=<DECIMAL>
	{
		try {
		value = Double.parseDouble(t.image);
		} catch (NumberFormatException nfe) {
		/* NOTE: this statement should never be reached because the
		*      token manager will only pass proper decimal numbers
	  *    to this code; however, Java requires a try-catch
	  *    clause in order to parse Strings into doubles */
```

```
                throw new ParseException("Invalid decimal number on line: " +
                t.endLine);
        }
        }
        | value = Number() )

        /* Return the parser result */
        { return value; }
}


/**
 * Parses a non-decimal number from a PCD file into a Java integer
 **
 * @return the coresponding Java int value
 */
int Number() :
{
        /* The integer value parsed by the function */
        int value = 0;

        /* The token to parse into an integer value */
        Token t = null;
}
{
        (
        /* Match the number token to parse */
        t=<NUMBER>

        /* Parse the token into a Java integer value */
        {
                try {
                value = Integer.parseInt(t.image);
                } catch (NumberFormatException nfe) {
                /* NOTE: this statement should never be reached because the
                *       token manager will only pass proper numbers to this
                *       code; however, Java requires a try-catch clause in
                *       order to parse Strings into integers */
                throw new ParseException("Invalid number on line: " +
                        t.endLine);
                }
        }
        )

        /* Return the parser result */
        { return value; }
}

/**
 * Parses a boolean token into a java boolean
 **
 * @return the value of the boolean
 */
boolean Bool () : {}
{
```

```
       /* Return true if match the T_TRUE token */
       ( <T_TRUE>  { return true;  }

       /* Return false if match the T_FALSE token */
       | <T_FALSE> { return false; } )
}


/**
 * Asserts indentation level (calls token_source.testIndent)
 **
 * @param scope the number of indents required
 */
void assertIndent (int scope) :
{}
{
       {
       if (!testIndent(scope)) {
       throw new ParseException("Indentation error on line: "
              + getToken(1).beginLine + " with an indentation of "
              + (token_source.getIndent() * token_source.INDENT_SIZE)
              + " spaces (expected "
        + (scope * token_source.INDENT_SIZE) + " spaces)");
    }
       }
}

/**
 * Tests indentation (NOTE: this calls the token manager)
 **
 * @param scope the number of indents required
 */
boolean testIndent (int scope) :
{}
{
       { getToken(1); }
       { return (token_source.getIndent() == scope && getToken(1).kind != EOF); }
}


/**
 * Matches new line characters including preceding whitespace
 */
void nl() : {}
{
       ( <WSP> )* ( <NL> | <EOF> )
}


/**************/
/* LEXER DATA */
/**************/

TOKEN_MGR_DECLS: {
       /**
```

```
       * Stores the current indentation scope
       */
       private int indent = 0 ;

       /**
       * Used to store the size of an indent in spaces
       * This is necessary for calculations within the Java program
       */
       public  static final int INDENT_SIZE = 4 ;

       /**
       * Returns the current indentation level
       **
       * @return the current indentation level
       */
       public int getIndent() {
       return indent;
       }
}

/* KEYWORD CLASSES */

/* PARAMETER TYPE KEYWORDS */
<*> TOKEN: {
             < T_BUTTON:   "button"   > : DATA
       | < T_CHOOSER:      "chooser"  > : DATA
       | < T_COMBOBOX:   "combobox" > : DATA
             | < T_DECIMAL:      "decimal"  > : DATA
             | < T_DIR:          "dir"        > : DATA
             | < T_FILE:         "file"       > : DATA
             | < T_LIST:         "list"       > : DATA
             | < T_NUMBER:       "number"   > : DATA
             | < T_TEXT:         "text"       > : DATA
             | < T_TEMPFILE:   "tempfile" > : DATA
             }

/* SYSTEM TYPE KEYWORDS */
<*> TOKEN: {
             < T_LINUX:    "linux"      > : DATA
             | < T_OSX:          "osx"        > : DATA
             | < T_SOLARIS:      "solaris"  > : DATA
             | < T_UNIX:         "unix"       > : DATA
             | < T_WINDOWS:      "windows"  > : DATA
             }

/* MACHINE ARCHETECTURE KEYWORDS */
<*> TOKEN: {
             < T_X86:      "x86"        >            : DATA
             | < T_AMD64: ("amd64" | "x86_64") > : DATA
             | < T_SPARC:        "sparc"      >            : DATA
             }

/* BOOLEAN KEYWORDS */
<*> TOKEN: {
             < T_FALSE:    "false"      > : DATA
```

```
              | < T_TRUE:         "true"       > : DATA
              }

/* FILE FORMAT TYPES */
<*> TOKEN: {
              < T_CSV:     "csv"        > : DATA
              | < T_TSV:          "tsv"        > : DATA
              | < T_FASTA:        "fasta"      > : DATA
              | < T_FLAT:         "flat"       > : DATA
              | < T_GDE:          "gde"        > : DATA
              | < T_GENBANK:      "genbank"   > : DATA
              | < T_RAW:          "raw"        > : DATA
              | < T_MASK:         "colormask" > : DATA
              }

/* FILE DIRECTION TYPES */
<*> TOKEN: {
              < T_IN:      "in"         > : DATA
              | < T_OUT:          "out"        > : DATA
              }

/* PARAMETER FIELD NAMES */
<*> TOKEN: {
              < T_CHECK:   "check"      > : DATA
              | < T_CHOICES:      "choices"   > : DATA
              | < T_CLOSE:        "close"      > : DATA
              | < T_CONTENT:      "content"   > : DATA
              | < T_DIRECTION:  "direction" > : DATA
              | < T_FORMAT:       "format"     > : DATA
              | < T_MAX:          "max"        > : DATA
              | < T_MIN:          "min"        > : DATA
              | < T_OVERWRITE:  "overwrite" > : DATA
              | < T_TYPE:         "type"       > : DATA
              | < T_SAVE:         "save"       > : DATA
              | < T_SHELL:        "shell"      > : DATA
              }

/* OPTIONS KEYWORDS */
<*> TOKEN: {
              < T_ICON:    "icon"       > : DATA
              | < T_SYSTEM:       "system"     > : DATA
              | < T_TIP:          "tip"        > : DATA
              }

/* CONTENT KEYWORDS */
<*> TOKEN: {
       < T_CANVAS:   "canvas"     > : DATA
       | < T_SELECTION:  "selection" > : DATA
              }

/* CODE KEYWORDS */
<*> TOKEN: {
              < T_AND:     "and"        > : DATA
              | < T_IF:           "if"         > : DATA
              | < T_OR:           "or"         > : DATA
```

```
              | < T_THEN:         "then"       > : DATA
              | < T_XOR:          "xor"        > : DATA
              }

/* MISC. KEYWORDS */
<*> TOKEN: {
              < T_ALL:      "all"        > : DATA
              | < T_DEFAULT:      "default"   > : DATA
              | < T_EXEC:         "exec"       > : DATA
              | < T_LABEL:        "label"      > : DATA
              | < T_ITEM:         "item"       > : DATA
              | < T_MENU:         "menu"       > : DATA
              | < T_CMDNAME:      "name"       > : DATA
              | < T_PARAM:        "var"        > : DATA
              | < T_TABSET:       "tabset"     > : DATA
              | < T_TAB:          "tab"        > : DATA
              | < T_PANEL:        "panel"      > : DATA
       | < COMMA:          ","          > : DATA
              }

/* DATA TOKENS */
<*> TOKEN: {
              < TEXT:      "\""( ~["\""] | <DOUBLEQ> )*"\""
       > : DATA
              | < DECIMAL: ( <NUMBER> "."  | <NUMBER> "." <DIGITS> | ( <SIGN> )? "."
<DIGITS> ) > : DATA
       | < NUMBER:  ( <SIGN> )? <DIGITS>
       > : DATA
              | < ID:      (["a"-"z","A"-"Z"])(["a"-"z","A"-"Z","_","0"-"9", "."])*
       > : DATA
              }

/* HANDLE NEW LINES */
<DATA> TOKEN: {
              < NL: ( <EOL>
                     | <COEL> )  >  { indent = 0; } : DEFAULT
       }

/* SKIP COMMENTS! */
<*> MORE:  { < COMMENT: ("#"(~["\n","\r"])+) > : DEFAULT }

<DATA> TOKEN: {
       < WSP: ( <SP>  | <TAB> )+  >
       }

<DEFAULT> SKIP: {
              < <EOL>  >          { indent = 0; }
              | < <COEL> >        { indent = 0; }
              | < <TAB>  >
              | < ( <SP> ){4}   > { indent++;   }
              | < ( <SP> ){1,3} >
              }


/* DATA SUPPORT TOKENS */
```

```
<*> MORE:  {
             < #SIGN:     "-"                 >
             | < #DIGITS:  (["0"-"9"  ])+     >
       | < #SP:     " "                  >
             | < #TAB:    "\t"                >
             | <  EOL:    "\r\n"
                    | "\n"
             | "\r"             >
             | < #COEL:   ( "#" (~["\n","\r"])+) >
             | < #DOUBLEQ:   "\"\""           >
             }
```

B. References