PCD Specification (update Aug. 21, 2012)

A. Background and definitions

In computer science, the syntax of a programming language is the set of rules that define the combinations of symbols that are considered to be correctly structured programs in that language. The syntax of a language defines its surface form.[1] Text-based programming languages are based on sequences of characters .

The lexical grammar of a textual language specifies how characters must be chunked into tokens. Other syntax rules specify the permissible sequences of these tokens and the process of assigning meaning to these token sequences is part of semantics. The syntactic analysis of source code usually entails the transformation of the linear sequence of tokens into a hierarchical syntax tree (abstract syntax trees are one convenient form of syntax tree). This process is called *parsing*, as it is in syntactic analysis in linguistics . Tools have been written that automatically generate parsers from a specification of a language grammar written in Backus-Naur form.

Syntax

The term "syntax" refers to the structure of a programming language, in particular, the different series of symbols and words that make up the basic parts of the language. The most common way of specifying the syntax of a language is use a notation known as Backus-Naur Form.

Backus-Naur Form

Terminals & Nonterminals

Backus-Naur Form, BNF for short, is a notation used to describe grammars. The notation breaks down the grammar into a series of rules - which are used to describe how the languages lexical and syntactic structures are used to form different logical units

The actual reserved words, symbols, etc... of the grammar are represented "terminals". In Backus-Naur Form, terminals are usually left without any special formatting or are simply delimited by single or double quotes. Examples include: if, while, '=' and identifier.

Syntactic rules are represented with a "nonterminal" - which are structure names. Typically, nonterminals are delimited by angle-brackets, but this is

not always the case. Examples include <statement> and <exp>. Both terminals and nonterminals are referred to generically as "symbols".

Productions

The actual syntax of the grammar is specified by combining terminals and nonterminals into syntactic rules known as "productions". They have the following format:

where *N* is a nonterminal and *s* is a series of zero or more terminals and nonterminals. Different alternatives can be specified in Backus-Naur Form. For readability, often productions are grouped together and separated by a “pipe” symbol - which is read as the word “or”.

Basically, a production has the following properties.

- The production starts with a single nonterminal, which is the name of the structure being defined
- This nonterminal is followed by a **::=** symbol which means “as defined as”. The ::= symbol is often used interchangeably with the symbol. They both have the same meaning.
- The symbol is followed by a series of terminals and nonterminals.

LALR

LALR parser (or **look-ahead LR parser**) is a type of LR parser (Left to Right) based on a finite-state-automata concept. The LALR algorithm is used to perform syntactic analysis for the parser..
LALR algorithm is a simple state transition graph. Each state represents a different stage the system can be in as it parses a string.
Each state represents a point in the parse process where a number of tokens have been read from the source and rules are in different states of completion. Each production in a state of completion is called a "configuration" and each state is really a configuration set.
For each token received from the. lexer, the LR algorithm can take four different actions: Shift, Reduce, Accept and Goto.
For each state, the LR algorithm checks the next token on the input queue against all tokens that expected at that stage of the parse. If the token is expected, it is "shifted". This action represents moving the cursor past the current token. The token is removed form the input queue and pushed onto the parse stack.

A reduce is performed when a rule is complete and ready to be replaced by the single nonterminal it represents. Essentially, the tokens that are part of the rule's handle - the right-hand side of the definition - are popped from the parse stack and replaced by the rule's nonterminal.
When a rule is reduced, the algorithm jumps to (gotos) the appropriate state representing the reduced nonterminal. This simulates the shifting of a nonterminal in the LR state machine.
Finally, when the start symbol (the nonterminal used to represent the entire grammar) is reduced, the input is both complete and correct. At this point, parsing terminates.

2. PCD Formal Syntax

The first step in creating BioPCD was to create a superset of the language referred to as PCD (Pythonesque Command Description). Much as XML is a superset of a large family of markup languages including HTML, PCD is a superset of BioPCD. PCD defines the core of terminals and high-level non-terminals required to describe a data of almost any type. BioPCD is adds to PCD the constructs necessary to describe GUI components and syntax for executing system commands.

2.1. Terminals (as regular expressions)

All of the below types in PCD are written as regular expressions. The format of these expressions are the same as they would be specified in Perl.

```
<bool>      ::= true
            ::= false
<text>      ::= ”([^”]|””)*"
<id>        ::= [^”]+
<number>    ::= [0-9]+\.?[0-9]*
<comment>   ::= #[^\n]*
```

2.2. Non-terminal productions

PCD code is defined in blocks as follows:

```
<block>     ::= <field> <indent+1> <block> <indent-1>
            ::= <field> <value>
            ::= <block> <indent> <block>
<field>     ::= <data>
<value>     ::= <data>
<data>      ::= <bool>
            ::= <text>
            ::= <id>
            ::= <number>
```

The above tags work similarly to Python, in that indentation defines the scope of parameters. Where this differs from Python is that the indentation is specified as four spaces per indent level. This prevents the mixing of tabs and spaces, which is a common problem in Python.

<indent> means to add a new line character and maintain the same indentation level, while <indent+1> indicates that the indentation of the line should be increased by one indentation unit, and <indent-1> indicates that the indentation of the line should be decreased by one indentation unit. Terminal symbols are separated by whitespace, and the only whitespace that is specified in the grammar below are new lines and their indentation effect. Comments may be interspersed at the end of any line.

3. Application of BioPCD - The BioLegato Menu Language

3.1 Working assumptions

BioPCD is designed around two working assumptions, to allow for flexibility. The source of the data is unspecified, but is assumed to be data selected elsewhere within an applcation. This makes BioPCD fit within a range of possible applications. It would even allow BioPCD to be implemented in the context of a web browser. The menu could be implemented as an application with a single standalone window, a panel within an application, or a pull-down menu.

For the purposes of this paper, we describe BioPCD within the context of a programmable GUI that we have developed called BioLegato. A full description of BioLegato will be published elsewhere. Briefly, BioLegato is Java application in which BioPCD menus are chosen from a pulldown list, as illustrated in Figure 1.

3.2

BioLegato is implemented in JavaCC, which does some semantics checking in its parser. The major differences are that every block is distinct in what it can contain, the only identifiers currently supported are predefined keywords, the only fields that are not keywords are those in the choices subsection, most of the fields must be specified in the order shown below (with the exception of panel, var, and table, which may be specified multiple times in any order, after the system parameter). The grammar is specified as follows (in a tree format showing the established hierarchy):

```
name
icon
tip
exec
system
var
      type
      label
      shell
```

```
      close
      save
      overwrite
      content
      canvas
      content
      min
      max
      default
      choices
            ..... ......
panel
      var
            ......
tabset
      tab “....”
            var
                  ......
```

For the above tags, the following tags would only be used once per file: `name. Icon, tip, exec, system`. These tags correspond to the name of the menu item, the icon for the menu item, the tooltip text for the menu item, and the execution command for the menu item, respectively. The `exec` tag is used only for BioPCD commands which do not require the user to enter any parameters. These commands are launched immediately when the user clicks on the menu item (the user is not prompted for any input).

The `tabset` tag is used to indicate that a tabbed pane be created, and a tab created for each child `tab` tag.

The `panel` tag is used to create a panel for housing widgets. Panels are used to place more than one widget on the same line.

The `system` tag is accompanied by a list of computing platforms on which the menu should appear. This tag is useful if a program to be called from the shell command is only available on some platforms but not others. By default the list is empty, meaning that the menu should appear on all platforms. If the list is not empty, then the menu will only appear on the specified platform. The set of tags corresponding to valid platform choices are implementation dependent. In the current version of BioLegato, the following system tags are allowed: `solaris-sparc; solaris-amd64; linux-intel; linux-x86_64; osx-x86_64; winxp-32; win7-64`.

The `var` tag is the most versatile of all of the tags presented thus far. The `var` tag is used to specify program parameters (or variables) for the most part. The only exception to this is program buttons, which use the var tag, but are not parameters. Rather, program buttons are used to run commands using the program parameters specified by other var tags. The var field is specified as follows:

```
var "source"
   type        chooser
   label       "Source"
   default     1
   choices
       "Commercial" "C"
       "All"       "A"
```

Note that each `var` tag has name associated with it. The name for a `var` tag is specified on the same line as the `var` tag. In the case above, the name of the var tag is *`source`*. This means that whenever `%source%` is encountered in an `exec` field or a `shell` field, it is replaced with whatever choice is selected in the variable. In the above example, the variable is a drop-down box with two options ("`Commercial`" and "`All`"). Therefore, if the user of BioLegato were to select "`Commercial`" in this program, and click a button to perform a command, each instance of `%source%` in the button's command would be replaced with "C" (the value of commercial).

Each `var` tag must have a different set of parameters in BioLegato, depending on what type of variable the `var` tag represents. For instance, using the chooser example above, each chooser variable must have the fields `type` and `choices`; type specifies what type of variable "`Source`" is, and `choices` specifies which `choices` should be available to the user. The fields `default` and `label` are optional fields, which specify the default value to select in the `chooser` (the first value the user will see selected before he or she changes it), and the text to display to the left of the `chooser` (this is so the user can understand what parameter the `chooser` is selecting for the program defined by the BioPCD menu).

The following list contains all of the var tag types currently supported in BioLegato. The parameters each var tag supports are listed below each type. The parameters in brackets [] are optional parameters. Each parameter below must be specified in the exact order that they are listed in each var type description (e.g. for buttons - type then label, then shell, then close):

Buttons:

```
   type button
 [ label "the text to display in the button" ]
   shell "the command to execute when the button is clicked"
 [ close false ]    # whether or not to close the parameter window
                    # when the button is clicked
```

Chooser, Comboboxes, or Lists:

```
   type chooser   # can also be combobox or list
 [ label "the text to display beside the list/combobx/chooser" ]
 [ default 0 ]    # specifies which choice should be selected by default
   choices
       "abc" "1"    # each line of the format "name" "value" specifies
                    # a choice within the list/combobox/chooser
                    # the name specifies what the user will see and select
```

```
                    # whereas the value will specify what the program
                    # sees from the user's selection
```

Text (textboxes):

```
   type text
 [ label "the text to display beside the textbox" ]
 [ default "the default text for the text" ]   # specifies what text should be
                                               # entered in the textbox by
default
```

Number:

```
   type number
   label "the text to display beside the number entry widget" ]
   min 0          # specifies the minimum number selectable by the number
widget
   max 10000      # specifies the maximum number selectable by the number
widget
 [ default 0 ]    # specifies the default number to be selected in the widget
```

File (an external file specified by the user):

```
   type file
 [ default "the default filename to have selected" ]
```

Directory (an external directory specified by the user):

```
   type dir
 [ default "the default directory path to have selected" ]
```

Tempfile (a temporary file containing all of the data selected in BioLegato's main canvas):

```
   type tempfile
   direction out  # direction can either be in or out
                  # the direction specifies whether the file will be read as
input
                  # for the program (in), or the file will receive output
from the
                  # program, which will be displayed/received by the canvas
(out)
   format fasta   # the file format represented by the tempfile
                  # currently BioLegato supports the following file format
options:
                  #     csv, tsv, fasta, flat, gde, genbank, raw, mask
 [ save true ]    # specifies whether the file should be "saved" or deleted
after
                  # program execution (true/false)
 [ overwrite true ]   # this parameter is only valid if the direction is out.
                      # this parameter determines whether the contents of the
                      # file will replace the current selection when the file
                      # is read back into the canvas
 [ content canvas ]   # this parameter is only valid if the direction is in.
                      # this parameter currently accepts two values:
```

```
                    #     canvas, selection
                    # this parameter determines whether the data that is
                    # written to the temp file is the current seletion
(selection)
                    # or the entire contents of BioLegato's main canvas
(canvas)
```

BioPCD is case-insensitive. This is because case-sensitivity is often confusing. People can simply forget whether something is upper or lower case. Also, case-sensitivity adds the potential for identical words, which only differ only in their case, to be defined differently multiple times, further adding to the confusion. Therefore, in attempts to avoid confusion for the user, we made PCD case-insensitive.