A Visual Basic Form Scripting Language for Blind Programmers

Amol Jain

August 29, 2004

1 Abstract

Blind programmers have been able to work productively along side their sighted peers in a command-line environment with the aid of tools such as screen readers and Braille terminals. Over the past few decades, however, computers have transitioned from text-based to graphical interfaces, and this change has resulted in the blind being unable to design commonly used applications. A scripting language was thus created that would allow blind programmers to design graphical Microsoft Visual Basic forms by specifying the layout in a text file rather than using the conventional "point and click" method. The compiler for the language was written in Microsoft Visual C++ and utilizes a table-driven parser to interpret the user's text file and output its respective form file. In addition to allowing blind programmers to create graphical forms, the language bears a strong resemblance to Visual Basic syntax and maintains Visual Basic's status as a RAD (Rapid Application Development) programming language.

2 Introduction

Computers have undergone considerable evolution from the time of their conception to the state in which they function today. Text-based computing, such as that of UNIX operating system, has largely been replaced by various graphical environments provided by X-Windows, Macintosh Operating System, and Microsoft Windows (Ceruzzi, 2000). This change has been accompanied by a great number of drawbacks for blind programmers, who previously were able to effectively compete with sighted professionals in their field.

The blind generally have a high rate of unemployment (Kirchner, 1997), but facilitated by screen readers, voice recognition programs and Braille terminals, blind programmers were able to hold jobs and develop main-stream applications. The shift towards Graphical User Interfaces (GUIs), however, has made programming, a profession in which large numbers of the blind could formerly participate (Siegfried, 2002), less feasible. Further complicating the issue is the fact that the development environments used to create GUIs often employ graphical tools themselves, thereby making them inaccessible to the blind (Sajka, 2004). A prime example of this is Microsoft Visual Basic.

Visual Basic became a popular tool for developing graphical programs because of its facile nature. All a user was required to do is drag and drop objects such as a command button or a text box onto a form and then write some code in order to make the application functional. This, however, presented a serious impediment to blind programmers, as they could not create graphical forms via the traditional "point and click" method that the majority of Visual Basic programmers used. The alternative to this would be to open a text editor and create a form file, which stores the locations and properties of the form and all the objects on it. This course would be similarly difficult because it would require knowledge of complicated Visual Basic form file syntax and the ability to generate extremely precise positional values for an aesthetically pleasing form.

It is under these circumstances that a Visual Basic form scripting language is proposed. The project, known as Molly, entails the construction of a compiler which parses a form scripting file and outputs a standard Visual Basic form file. Essentially, a user would be able to design the layout the desired form through a simple text file and create a graphical form after running it through the compiler. The language's syntactical similarity with Visual Basic along with its entirely text-based interface would make it an ideal tool for the blind to venture into the arena of GUI programming.

In its current state, the language is still prototypical, though several developments have been made contributing to its maturation. The first of these developments is the addition of more objects. Initially, the language only supported the placement of 5 basic objects on the form: command buttons, comboboxes, frames, checkboxes, and textboxes. Along with a greater number of objects are also modifications to existing objects that allow for more user input, thus resulting in more flexible and customizable forms. Finally, a refinement of the methods used to place objects in their respective locations will give way to better looking forms.

The Molly project intends to increase the accessibility of the graphical Visual Basic programming environment to blind programmers by allowing them to design forms and create forms through a text-based form scripting language.

3 Development

Development of the Molly compiler took place in a Windows 2000 SP3 environment with Intel Pentium 4 processors. The compiler was written in Microsoft Visual C++ 6.0 and the graphical forms produced were viewed and analyzed with Microsoft Visual Basic 6.0, also part of the Visual Studio 6.0 suite of applications.

The source code for Molly was divided into 7 files, totaling 4185 lines: molly.cpp (897), ast.cpp (1749), ast.h (271), symbol.cpp (623), symbol.h (198), scan.cpp (405), and scan.h (42).

3.1 molly.cpp

The molly.cpp file is the main component of the compiler as it contains the code for the parser. Originally, the compiler employed a recursive descent parser in which there were a great number of Visual Basic object specific functions. One function would call another function, which would call yet another function, and the process would continue in a recursive manner. Although relatively facile in implementation, the recursive descent parser did not have any centralized location at which one could examine all the productions involved in parsing the syntax of the molly form scripting language. Thus, the new version of molly.cpp contains a table driven parser, a cleaner alternative. The main information necessary for parsing is now contained in a production table, a production index, and a production array. The production table is essentially a matrix whose rows are non-terminals and whose columns are tokens, or key words. The numbers it contains are all references to elements of the production index, which references the actual productions within the production array. The following example is taken from the production array and is the production for a command button:

```
{Term, tokcmdbutton},          {Action, AcSetThisSon},
{Term, tokid},                 {Action, AcInstallCmdButton},
{Nonterm, NTReturns},          {Nonterm, NTCaptionAttrib},
{Action, AcGoToFather},        {Term, tokend},
{Nonterm, NTReturns}
```

The productions are clearly represented in the table driven parser. In this example, the command button token, “commandbutton”, is required in the input file, after which the program creates a new node in the abstract syntax tree to store the relevant data. Next, an identifier for the command button is expected, and a function is called to annotate the various properties of the command button. After a carriage return, the caption attribute of the object is set, and the program shifts focus to the father of the current node. Finally, the token “end” is expected in the input file to complete the declaration of a command button in the molly scripting language. All the other productions are laid out in a similar manner in molly.cpp.

3.2 ast.cpp and ast.h

The ast files manage the abstract syntax tree, which is essentially a large data structure that holds the information from the parsed input file within nodes. The functions that annotate, or add all the object specific properties to nodes, reside within ast.cpp. Furthermore, these files handle the translation of the abstract syntax tree into Visual Basic form file syntax and are thus in charge of the actual creation of the graphical form file.

3.3 symbol.cpp and symbol.h

The symbol files contain an enumeration of all the tokens in the custom defined tokentype form and in string form for referencing. They also creates hash codes for all the tokens which are searched to see if the token currently being examined by the program is valid. Finally, symbol.cpp contains various functions that manage the attribute table.

3.4 scan.cpp and scan.h

The scan files handle the lexical analysis of the input file. They open and close the file specified for reading and go through it character by character. The scanner, upon reading a word, number, or symbol, sends the information in the form of a token to the parser, where it is processed as part of a production. The scan files used in the Molly project are modified versions of those originally written by Dr. Siegfried for the JASON compiler.

4 Forms in Microsoft Visual Basic

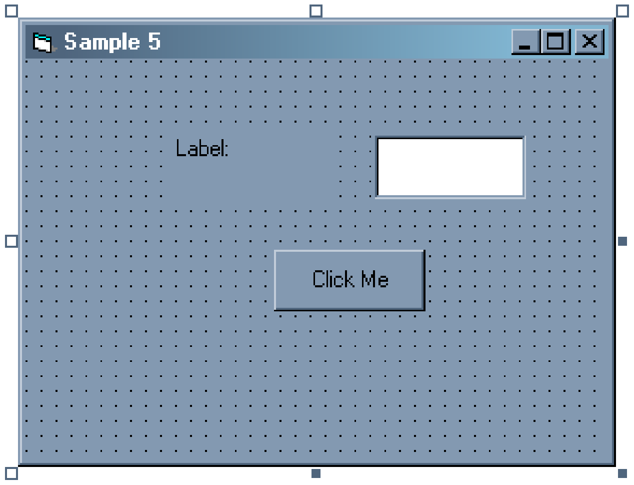

Figure 1: A Simple Form

Graphical forms in Visual Basic are created by clicking on objects in a toolbar and then dragging them onto a form. Once on a form, objects may be resized by clicking on any of their corners and dragging the pointer until the desired size is achieved. Repeating this process for several objects allows one to create a graphical interface such as the one displayed in Figure 1.

When a form is saved in Visual Basic, the .frm extension is appended to the filename. If the form file is double clicked or opened normally, the computer will load an instance of Visual Basic, and the form will be displayed graphically in the same state in which it was previously saved. If the same file is opened in a text editor, however, it is revealed that all the information is actually stored in text format. An example can be found in Appendix A, which is the text version of Figure 1. The file is organized into groups of objects, their properties, and positional values. The location of an object is determined by its top and left property, which are specified in twips, or twentieths of a point. While modifying these values once set might be easy, generating them from scratch to produce a well-laid out form is not. The Molly scripting language simplifies the process of creating one of these text files, and thus a graphical form, by not requiring the user to designate specific positional values or memorize the form file syntax.

5 The Scripting Language

5.1 Invoking the Compiler

The Molly compiler, named molly.exe, is a text-based application that is run from a command prompt. The user creates a form script, which contains the description of the desired graphical form, and saves the text file with the .fms extension. The script can then be compiled by the following command:

```
molly filename.fms
```

An alternative would be to run molly.exe without any argument. The user would then be prompted to input the file name, and compilation would proceed as above. If the form script contains errors, compilation will halt, and the user will be presented the relevant error output, such as the line in the form script where the error was found. If compilation is successful, a Visual Basic form file with the .frm extension will appear in the same directory. If the example above completed compiling, the file ***filename*.frm** would be created.

5.2 Form Script Syntax

The syntax of the Molly scripting language was designed to be as similar as possible to that of Visual Basic, as to avoid having the user learn an entirely new language. Visual Basic and the form scripts are similar in that both are case insensitive with free form lines that end with a carriage return. Comments in both languages are also specified with apostrophes and continue until the end of the line.

All form script files share a similar structure that can be generalized as follows:

```
1 Form frmName
2 Location = VerticalAttribute HorizontalAttribute
3 Caption = "Caption on Title Bar of Form"
4 Organization = Rows or Columns
5 Section
6 ObjectDeclarations
7 End ' This is a comment
8 ...
9 End ' End of Form
```

The first line declares a new form and the form name. The second line specifies where the form will appear when the Visual Basic application is run. The *VerticalAttribute* may be either `top`, `middle`, or `bottom`. Likewise, the *HorizontalAttribute* may be either `left`, `center`, or `right`.

The form may be organized by either rows or columns. If the organization is rows, the first object is placed in the top left corner of the form, and the next object is placed to its right until a new section is declared. Objects in the new section will be placed in the same fashion only in a lower row.

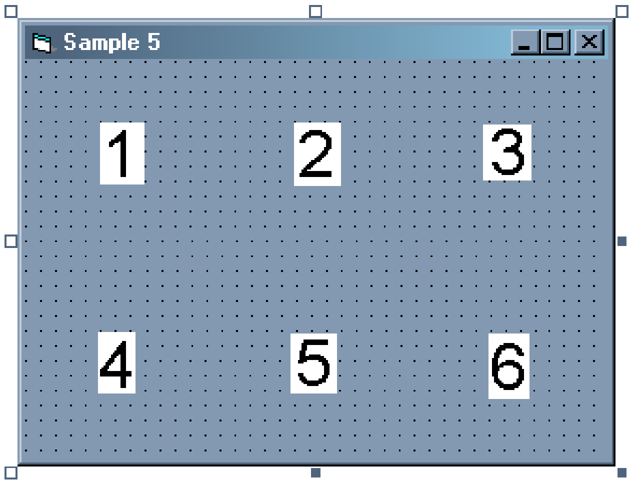

Figure 2: Organization Rows

If the organization is columns, the first object is also placed in the top left corner of the form, but the next object is placed below it until a new section is declared. Objects in the new section will be placed similarly in a new column to the right.

A section requires at least one object declaration. There may be any number of objects within a section and any number of sections within a form as long as the form's

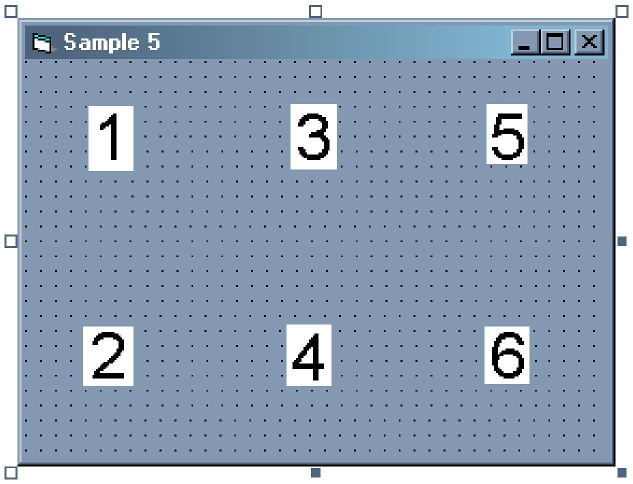

Figure 3: Organization Columns

height does not exceed 11520 twips and its width is not greater that 15360 twips. All sections are closed with an End, as is the entire form.

5.3 Declaring Objects

The Molly scripting language originally supported only 5 basic objects: command buttons, comboboxes, frames, checkboxes, and textboxes. Currently, the user is able to specify 6 additional objects: listboxes, timers, filelistboxes, drivelistboxes, directorylistboxes, and scrollbars.

An object is declared by writing the object type followed by unique identifier that conforms to Microsoft's naming conventions. The properties of the object are listed thereafter along with an End to complete the specification.

The syntax for all objects may be derived from the BNF Grammar listed in Appendix B. To declare a checkbox, for instance, the following line in the grammar must be examined:

$\langle \textit{CheckBoxDeclaration} \rangle$ ::= `checkbox id` $\langle \textit{Returns} \rangle$ $\langle \textit{CaptionAttribute} \rangle$
$\langle \textit{SizeAttributes} \rangle$ `end` $\langle \textit{Returns} \rangle$

First, the terminal `checkbox` is required along with a unique identifier. Next is the nonterminal *Returns*. The definition for *Returns* is as follows:

$\langle \textit{Returns} \rangle$ ::= $\langle \textit{Returns} \rangle$ $\langle \textit{cr} \rangle$ | $\langle \textit{cr} \rangle$

There are two possible definitions. One is simply a single carriage return, and the other is a carriage return along with another *Returns*. This recursively translates to any number of carriage returns. Thus, after the terminal `checkbox` and an identifier, at least one carriage return is expected.

Next expected is the *CaptionAttribute*:

$\langle \textit{CaptionAttribute} \rangle$ ::= `caption` = $\langle \textit{String} \rangle$ $\langle \textit{Returns} \rangle$

Next, the definition of *String* would be looked up in the grammar, and this procession would continue until there are no nonterminals left to translate. An example checkbox declaration would therefore be something like:

```
CheckBox chkSamp
        Caption = "This is a Checkbox!"
        Height = 3
        Width = Large
End
```

One would follow its respective production in the grammar to declare any other object.

5.4 Form Script for Figure 1

```
Form sample5
Location = Top Right
Caption = "Sample 5"
Organization = Columns

Section
        TextBox txtSamp
                Height = 2
                Width = Small
                Label = "Label: "
        End
        CommandButton cmdSamp
                Caption = "Click Me"
        End
End ' Section

End ' Form
```

The above is the form script for the form displayed in Figure 1. It follows the general structure of forms as listed above, and contains 2 object declarations, one for a textbox, and the other for a command button. On the form, however, there are actually 3 objects because the scripting language was designed such that textboxes have a label property which creates a label to the left of the textbox.

This form script is also the equivalent of the .frm text file in Appendix A. The script in the Molly language is 17 lines as compared to the 37 lines required to directly write a Visual Basic Form file and is more readable as well.

6 Future Developments

Although the Molly scripting language and compiler have significantly improved over the past few months, there is still much room for future development and improvement. The language currently supports using 11 objects, but this still not much compared to the number that Visual Basic makes available. Support for additional objects such as shapes, picture boxes, database controls, and ActiveX controls are possible improvements.

At the moment, a user is required to specify all properties of a certain object in the order in which they are listed in the production array and grammar in Appendix B. This is inconvenient as it necessitates that the user memorize the order for every object or consult the documentation frequently. Future versions of Molly will allow the properties to be specified in any order. Even in this situation, however, the user must remember all of an object's properties in order to declare it. Thus, a set of default property values will be created for use in the case that the user does not provide certain properties.

As expansive as the Molly scripting language may become, it is difficult to foresee all the object properties a user may wish to implement. The language can be made much more flexible by allowing the user to write standard Visual Basic code within the form script to specify properties not implemented in the scripting language.

The Molly compiler was designed for use with Microsoft Visual Studio 6, and creates .frm files which correspond to Visual Basic 6. The most recent version of Visual Studio, however, is Visual Studio .NET, and the text files with which it stores forms uses a different syntax. Visual Basic .NET contains a utility for converting Visual

Basic 6 forms into those which are .NET compliant, but it would be inconvenient for blind programmers to use it. The Molly scripting language will remain essentially the same, but support for Visual Basic .NET will require writing another back-end for the compiler. The user would then be presented with a choice between producing a Visual Basic 6 or Visual Basic .NET form upon running the Molly executable. An alternative would be to provide an argument additional to the form script file specifying the type of form desired when running the compiler.

Finally, the Molly compiler still has not been tested by the blind programmer community. Testing is one of the first things Molly's developers intend to have done before continuing with improvements upon the program.

Information, including the compiler and documentation, may be found at http://www.adelphi.edu/ siegfrir/molly.

References

Ceruzzi, P. E. (2000). *A History of Modern Computing*. The MIT Press.

Kirchner, C., & Schneidler, E. (Sept/Oct 1997). *Journal of Visual Impairment and Blindness*.

Sajka, J. (Director of Information Technology for the American Federation for the Blind). (2004). Interview with U. Obianyo-Agu.

Siegfried, R. M. (2002). A Scripting Language to Help the Blind to Program Visually. *ACM SIGPLAN Notices*, 32(2), 53-56.

Appendix A

Figure 1 in Text Format

```
VERSION 5.00
Begin VB.Form sample5
   Caption         = "Sample 5"
   ClientHeight    = 3195
   ClientLeft      = 60
   ClientTop       = 345
   ClientWidth     = 4680
   LinkTopic       = "Form1"
   ScaleHeight     = 3195
   ScaleWidth      = 4680
   StartUpPosition = 3  'Windows Default
   Begin VB.TextBox txtsamp
      Height          =   510
      Left            =   2800
      MultiLine       =   1
      TabIndex        =   0
      Text            =   ""
      Top             =   600
      Width           =   1215
   End
   Begin VB.Label lbl1
      Caption         =   "Label: "
      Height          =   495
      Left            =   1200
      TabIndex        =   1
      Top             =   600
      Width           =   1215
   End
   Begin VB.CommandButton cmdsamp
      Caption         =   "Click Me"
      Height          =   495
      Left            =   2000
      TabIndex        =   2
      Top             =   1510
      Width           =   1215
   End
End
```

Appendix B

A BNF Grammar for the Molly Scripting Language

⟨*Form*⟩ ::= ⟨*Header*⟩ ⟨*FormAttributes*⟩ ⟨*OrgAttributes*⟩ ⟨*SectionDeclarations*⟩
`end`

⟨*Header*⟩ ::= `form` `constant` ⟨*Returns*⟩

⟨*FormAttributes*⟩ ::= ⟨*LocationAttribute*⟩ ⟨*CaptionAttribute*⟩

⟨*LocationAttribute*⟩ ::= `location` = ⟨*VerticalAttribute*⟩ ⟨*HorizontalAttribute*⟩
⟨*Returns*⟩

⟨*VerticalAttribute*⟩ ::= `top` | `middle` | `bottom`

⟨*HorizontalAttribute*⟩ ::= `left` | `center` | `right`

⟨*CaptionAttribute*⟩ ::= `caption` = ⟨*String*⟩ ⟨*Returns*⟩

⟨*OrgAttributes*⟩ ::= `organization` = ⟨*SectionOrg*⟩ ⟨*Returns*⟩

⟨*SectionOrg*⟩ ::= `rows` | `columns`

⟨*SectionDeclarations*⟩ ::= ⟨*SectionDeclaration*⟩ ⟨*MoreSectionDeclarations*⟩

⟨*SectionDeclaration*⟩ ::= `section` ⟨*Returns*⟩ ⟨*ObjectDeclarations*⟩ `end` ⟨*Returns*⟩

⟨*MoreSectionDeclarations*⟩ ::= ⟨*SectionDeclaration*⟩ ⟨*MoreSectionDeclarations*⟩ | ⟨*nil*⟩

⟨*ObjectDeclarations*⟩ ::= ⟨*ObjectDeclaration*⟩ ⟨*MoreObjectDeclarations*⟩

⟨*ObjectDeclaration*⟩ ::= ⟨*CommandButtonDeclaration*⟩ | ⟨*TextBoxDeclaration*⟩
| ⟨*ComboBoxDeclaration*⟩ | ⟨*FrameDeclaration*⟩ | ⟨*CheckBoxDeclaration*⟩
| ⟨*ListBoxDeclaration*⟩ | ⟨*TimerDeclaration*⟩ | ⟨*FileListBoxDeclaration*⟩
| ⟨*DriveListBoxDeclaration*⟩ | ⟨*DirectoryListBoxDeclaration*⟩
| ⟨*ScrollBarDeclaration*⟩

⟨*MoreObjectDeclarations*⟩ ::= ⟨*ObjectDeclaration*⟩ ⟨*MoreObjectDeclarations*⟩ | ⟨*nil*⟩

⟨*CommandButtonDeclaration*⟩ ::= `commandbutton` `id` ⟨*Returns*⟩
⟨*CaptionAttribute*⟩ ⟨*Returns*⟩ `end` ⟨*Returns*⟩

⟨*TextBoxDeclaration*⟩ ::= `textbox` `id` ⟨*Returns*⟩ ⟨*SizeAttributes*⟩ ⟨*LabelAttribute*⟩
`end` ⟨*Returns*⟩

⟨*SizeAttributes*⟩ ::= ⟨*HeightAttribute*⟩ ⟨*WidthAttribute*⟩

⟨*HeightAttribute*⟩ ::= `height` = ⟨*Number*⟩ ⟨*Returns*⟩

⟨*WidthAttribute*⟩ := `width` = ⟨*Size*⟩ ⟨*Returns*⟩

⟨*Size*⟩ ::= `small` | `medium` | `large`

⟨*LabelAttribute*⟩ ::= `label` = ⟨*String*⟩ ⟨*Returns*⟩

⟨*ComboBoxDeclaration*⟩ ::= `combobox id` ⟨*Returns*⟩ ⟨*SortedAttribute*⟩
⟨*StyleAttribute*⟩ ⟨*WidthAttribute*⟩ `end` ⟨*Returns*⟩

⟨*FrameDeclaration*⟩ ::= `frame id` ⟨*Returns*⟩ ⟨*CaptionAttributes*⟩
⟨*OptionDeclarations*⟩ `end` ⟨*Returns*⟩

⟨*OptionDeclarations*⟩ ::= `optionbutton id` ⟨*Returns*⟩ ⟨*CaptionAttribute*⟩
⟨*VisibleAttribute*⟩ `end` ⟨*Returns*⟩

⟨*VisibleAttribute*⟩ ::= `visible` = ⟨*Boolean*⟩ ⟨*Returns*⟩

⟨*Boolean*⟩ ::= `true` | `false`

⟨*CheckBoxDeclaration*⟩ ::= `checkbox id` ⟨*Returns*⟩ ⟨*CaptionAttribute*⟩
⟨*SizeAttributes*⟩ `end` ⟨*Returns*⟩

⟨*ListBoxDeclaration*⟩ ::= `listbox id` ⟨*Returns*⟩ ⟨*SortedAttribute*⟩ ⟨*StyleAttribute*⟩
⟨*SizeAttribute*⟩ ⟨*ColumnsAttribute*⟩ `end` ⟨*Returns*⟩

⟨*SortedAttribute*⟩ ::= `sorted` = ⟨*Boolean*⟩ ⟨*Returns*⟩

⟨*StyleAttribute*⟩ ::= `style` = ⟨*Number*⟩ ⟨*Returns*⟩

⟨*TimerDeclaration*⟩ ::= `timer id` ⟨*Returns*⟩ ⟨*IntervalAttribute*⟩ `end` ⟨*Returns*⟩

⟨*IntervaAttribute*⟩ ::= `interval` = ⟨*Number*⟩ ⟨*Returns*⟩

⟨*FileListBoxDeclaration*⟩ ::= `filelistbox id` ⟨*Returns*⟩ ⟨*SizeAttributes*⟩ `end`
⟨*Returns*⟩

⟨*DirListBoxDeclaration*⟩ ::= `dirlistbox id` ⟨*Returns*⟩ ⟨*SizeAttributes*⟩ `end` ⟨*Returns*⟩

⟨*DriveListBoxDeclaration*⟩ ::= `drivelistbox id` ⟨*Returns*⟩ `end` ⟨*Returns*⟩

⟨*ScrollBarDeclaration*⟩ ::= `scrollbar id` ⟨*Returns*⟩ ⟨*OrientationAttribute*⟩
⟨*LengthAttribute*⟩ ⟨*ScrollBarAttributes*⟩ `end` ⟨*Returns*⟩

⟨*OrientationAttribute*⟩ ::= `orientation` = ⟨*OrientType*⟩ ⟨*Returns*⟩

⟨*OrientType*⟩ ::= `horizontal` | `vertical`

⟨*LengthAttribute*⟩ ::= `length` = ⟨*Size*⟩ `Returns`

⟨*ScrollBarAttributes*⟩ ::= ⟨*MinAttribute*⟩ ⟨*MaxAttribute*⟩ ⟨*ValueAttribute*⟩
⟨*ChangeAttributes*⟩

⟨*MinAttribute*⟩ ::= `min` = ⟨*Number*⟩ ⟨*Returns*⟩

$\langle \textit{MaxAttribute} \rangle$ ::= `max` = $\langle \textit{Number} \rangle$ $\langle \textit{Returns} \rangle$

$\langle \textit{ValueAttribute} \rangle$ ::= `value` = $\langle \textit{Number} \rangle$ $\langle \textit{Returns} \rangle$

$\langle \textit{ChangeAttributes} \rangle$ ::= $\langle \textit{SmallChangeAttribute} \rangle$ $\langle \textit{LargeChangeAttribute} \rangle$

$\langle \textit{SmallChangeAttribute} \rangle$ ::= `smallchange` = $\langle \textit{Number} \rangle$ $\langle \textit{Returns} \rangle$

$\langle \textit{LargeChangeAttribute} \rangle$ ::= `largechange` = $\langle \textit{Number} \rangle$ $\langle \textit{Returns} \rangle$

$\langle \textit{Returns} \rangle$ ::= $\langle \textit{Returns} \rangle$ $\langle \textit{cr} \rangle$ | $\langle \textit{cr} \rangle$

$\langle \textit{String} \rangle$ ::= $\langle \textit{AlphaNumeric} \rangle$*

$\langle \textit{AlphaNumeric} \rangle$ ::= $\langle \textit{Letter} \rangle$ | $\langle \textit{Digit} \rangle$

$\langle \textit{Number} \rangle$ ::= $\langle \textit{Digit} \rangle$ $\langle \textit{Digit} \rangle$*

$\langle \textit{Letter} \rangle$ ::= `A` | `B` | ... | `Y` | `Z` | `a` | `b` | ... | `y` | `z`

$\langle \textit{Digit} \rangle$ ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`