**UNIVERSITY OF PUERTO RICO**

**MAYAGUEZ CAMPUS**

**COLLEGE OF ENGINEERING**

**COMPUTER ENGINEERING DEPARTMENT**

**Porygon Programming Language**

**Final Report**

**Jennifer A. Dobyns**

**Orlando Ruiz López**

**Jady Urbina Cima de Villa**

**ICOM4036 Section 036**

**Wilson Rivera**

**December/13/2016**

**Introduction**

Geometry is a common subject that a large amount of students and professionals alike have trouble with, mostly due to the tedious need to memorize a large number formulas that develop such a broad subject and, also, due to the potential requirement to solve long and complex calculations. These issues serve as the main motivation for the creation of Porygon. The Porygon project is a programming language that will make it easy to apply formulas when necessary and to solve long calculations in a fast, efficient and user-friendly way. The user will be able to insert partial data of a geometric shape to then find the remaining properties of the constructed shape and determine properties of a certain shape. The properties of the shapes will vary depending on the type of shape, while some general properties such as area will be included in every shape. With Porygon, most of geometry’s calculations will be just a line of code away.

**Tutorial**

<https://www.youtube.com/watch?v=z3DDAoz8PZk>

Click the link above to watch a video demonstrating the Porygon language’s basic features.

1. Before porygon can be used ensure that you have the latest version of java installed. You can find the latest version of java and how to download it at this location: [https://www.java.com/](https://www.java.com/en/)
2. Next open the command prompt and navigate to the installation folder.
3. Run this command in the command prompt:  **java -jar porygon.jar** will start a session with the Porygon translator.
4. If the command terminal shows **porygon>** that means that everything has worked properly and you can now start coding with Porygon!
5. Porygon is meant to build geometric shapes and perform geometric equations on them. Complex shapes or datatypes can be built using simpler shapes/datatypes such as two points forming a line. Here are a few commands you can run to see how this works:
   1. point pt : (1, 2)
   2. point pt2 : (3,4)
   3. line ln : (pt, pt2)
   4. distance(ln)
6. Notice that when declaring variables you specify the datatype and you use the : to assign a value to the variable, whereas with functions you simply use parenthesis with an existing variable between them. Functions only take variables that have already been declared, they do not take hard coded numbers:
   1. **print(1,2)** - this is wrong, because no variable is given.
   2. **print(pt)** - as long as pt has been declared this will print the values of pt.

**Reference Manual**

|  |  |
| --- | --- |
| **Variable declarations** |  |
| point *name* : (*x, y*) | Declares a variable of type point with the given x value and y value. |
| line *name* : (*x1, y1, x2, y2*)  line *name* : (*point1, point2*) | Declares a variable of type line with either 4 integers or with two points. |
| triangle *name* : (*x1, y1, x2, y2, x3, y3*)  triangle *name* : (*point1, point2, point3*) | Declares a variable of type triangle with either 6 integers or with three points. |
| quadrilateral *name* : (*x1, y1, x2, y2, x3, y3, x4, y4*)  quadrilateral *name* : (*point1, point2, point3, point4*) | Declares a variable of type quadrilateral with either 8 integers or with four points. |
| circle *name* : (*x1, y1, x2*)  circle *name* : (*point1, point2*)  circle *name* : (*line1*) | Declares a variable of type circle with either 2 integers, two points or a line which will work as the radius. |

|  |  |
| --- | --- |
| **Functions** |  |
| help | Displays all available commands with their respective descriptions. |
| print (*variable name*)  printall | Prints the values of the given variable or of every variable in the system. |
| distance (*line*) | Gives the distance between two points of a line. |
| area (*triangle*)  area (*quadrilateral*)  area (*circle*) | Gives the area of the given shape. |
| circumference (*circle*) | Gives the circumference of the given circle. |
| exit | Terminates Porygon. |

**Development**

**Architecture**



**Description of interfaces between modules**

In the Porygon language, the input is entered in through the standard input one line at a time. Since Porygon is built on the java language, the execution of the command is implemented through the main function of project. The code is saved as a string variable with in the main function

and passed through the tokenizer class.

The tokenizer class then takes the code and breaks it into tokens and saves them in a list. Since the tokenizer is an object this list of tokens can be accessed later in the main function.

Next the main function creates a list of parsers, one for each type of command. Each parser contains a function that has a regular expression that represents the syntax of the command that will trigger that parser. The main function compares the line of code with each regular expression and when one matches it creates a new “block” for the code using the specific parser for it.

The block contains unique values and methods depending on which type of block it is. For example the variable blocks store the values that the user assigned to the variable. The main function then uses the block to execute the command corresponding to the type of block it is.

**Software development environment**

* Eclipse - The IDE that we used to develop porygon is eclipse since it something that we were used to using and it is commonly used to develop.
* GitHub - Git was used as the version control of our code.
* Pogo - This is a language that we found that was developed as a tutorial and so we used this as a framework for our language <https://github.com/pogostick29dev/Pro>

**Testing Environment**

While developing Porygon the team did continuous testing by using the “Test Early, Test Often” (TETO) motto to make sure every new piece of code was working well, that it was well connected with the previous code and that it didn’t affect in a negative way.To test the translator for the Porygon language, the team used the Eclipse Java Integrated Development Environment (IDE) since we had previous experience with it. After the translator was completely done, the team used the Windows’ Command Prompt to verify that the code would run smoothly from a Command Line without the need of Eclipse.

**Conclusions**

During the start of the development we were planning on using a parser called JavaCC, however we discovered that there weren’t very many support documents for the current version so trying to get it to work would be more work than it was worth. Eventually we found a language that someone had developed as a tutorial for other people to create their own language from scratch. While this language used concepts more complex than what we wanted our language to accomplish, we were able to modify it so that it would run our language the way we wanted. Next we spent time adding our own datatypes and commands for Porygon. Learning how to use the object oriented programming of java in order to create a point variable proved to be challenging, but once we learned how to do this, adding more complex datatypes was easier. We also kept modifying the regular expressions in the parsers whenever we realized we wanted our code to be read in a different way.