Final Report for Pioneer DSL Class of 2018

Yufan Liu
Friends' Central School
hliu@friendscentral.org

This paper is intended to wrap up the lesson from Domain-Specific Language and report the individual project: Geometry-Draw. Geometry-Draw is an external DSL with the implementation in Scala. All works are done by myself.

1 Introduction

This work is intended to provide people a platform to liberate their creativity. It is interesting in the way of its creativity and uncertainty. Say if someone wants to draw a bisector for a given angle, there are multiple ways to do it. However, simply using pens and scratch papers would create a lot of waste and isn't very convenient. In order to provide a better way for those geometry fanatics to try out their unique approaches, I decided to create this DSL. Geometry-draw can follow each step given by the fans and give an imagery result. Compared to the traditional way of drawing, computer-based drawing can make the work easier by just typing commands in daily-use language. This language is mostly for those geometry fans as well as math leaners. It is useful because it liberates users' thinking: users can command whatever they want in various ways to solve the problem. For math leaners, they can view how each step they call make difference on the canvas. For geometry fans, they can challenge themselves to solve the problem in various ways. This language does not need any programming experiences. The only thing required for this language is enthusiasm. There are currently very few DSLs for this domain. Wolfram-alpha and MATLAB are two of the most famous ones. They are both powerful tools for math problems as well as science. However, geometry-draw is considered more domain specific which only supports basic geometry drawing commands but has a better user experience.

2 Background

According to a research made in 2014 by IDC, 18.5 million people knew at least one programming language, which made 0.3% of the earth's total population[1]. With the development of the technology, the computer is gradually transforming the way people work in many fields: mathematicians are using the computer to prove hypothesizes, scientists are using the computer to reckon their discovery, and financial tycoons are using the computer to calculate complex financial models. Thinking back to the day when the early generation learned the programming language: C, C++, Java, they would hardly believe the data showed above. The program is complex and hard that it requires a bunch of time to fully manipulate one language. So who opened the door of programming to the world? Domain Specific Languages (DSLs).

DSL, Domain Specific Language, is a language designed for the specific purpose within the certain field, or domain. Compared to those most common languages, DSL only contains limited features while developed to have comparatively simple expressions, in turn, provides a friendlier user experience. Though for most of the time our consciousness tells us the programming languages are complex, most of the languages existing in the world, including SQR, can be defined as DSL. In many cases, experts in certain fields needed to use the computer to simplify their daily work would choose to learn the special DSLs designed for their expertise. With that in mind, the potential market for DSL is untold. It is worth spending time digging into the DSLs and uncovering its mysterious veil.

The implementation of the DSL will improve the efficiency for those domain experts, the people working in a certain domain with limited programming knowledge. For them, DSL is a strong and powerful tool to use. There are several occasions when DSL is perfectly suitable to implement. First, the user, or the domain expert, doesn't know how to program. The reason for developing the DSL is for the computer to understand domain experts' needs while programmer certainly does not need. Second, the code's clearness. DSL code needs to be clear in a way which has to be easy to read as well as less redundancy. If the DSL code is hard for other domain experts to read, such DSL has no difference with the General Purpose Language (GPL). Third, the code has to be as much like as Nature Language. Unlike GPL, which runs a program with data and operations, DSL runs a program with nouns and verbs. Because most DSLs are descriptive[2], in order to improve its expressive ability, it

[1]"How many people on earth know computer programming?", Quora. Accessed August 10, 2018. *https://www.quora.com/How-many-people-on-earth-know-computer-programming.*

[2]"Domain-Specific Language." Wikipedia. June 19, 2018. Accessed June 20, 2018. *https://en.wikipedia.org/wiki/Domain-specific_language.*

is necessary for language implementer to make the DSL as much alike nature language like English as possible.

In developing a DSL, the first thing languages implementers will do is go to a domain expert and try to understand the professional terminology and the way expert constructs a command. Implementer needs to add fluency to the DSL so that the nouns and verbs fit together. Then implementer will decide whether to use internal or external DSL. The difference between the two forms of DSLs is whether it has a host language. If the DSL is internal, it is necessary for the implementer to remove the host flavor. In Scala example, implementers most likely choose to use the "chaining in OOP" technique to remove the host flavor, as well as omitting the "." from method calls. Other techniques including infix operators, (re)defining operators, pre- and postfix operators and literal extension [3]. Unlike internal DSL, which is built on top of the host language, external DSL has its own grammar and operators. A good example of external DSL is Graphviz:

```
Graph {
  a -- b;
  b -- c;
  a -- c;
  d -- c;
  e -- c;
  e -- a;
}
```

it does not like any existing GPL while the code, like all other DSLs, is easy to understand. One thing language implementer needs to keep in mind is that DSL has to get as much close as possible to the domain problems, to eliminate unnecessary indirection and complexity[4]. The output of the above sample code is here. A well-designed DSL has to contain a rational and proper grammatical form. Take ContextFree as an example, this language covers expressions limited in generating certain pattern images. One example code:

```
shape Blossom {
 CIRCLE [size 2]
 loop 6 [r 60]
 LeftOrRightFlower [size 0.4 y 3]
}
```

[5] turns out to be easily understandable even without knowing anything about this language in prior. The grammar and function names are all daily words like circle, square and curve. These features make ContextFree a good tool for domain experts.

As mentioned before, despite the fact that the idea of DSL does not have much heat based on Google search, the potential of DSL is great. DSL can be seen as a bridge which connects implementers and domain experts. In that way, the domain experts can best use their ability to solve problems in their familiar language.

[3] *https://pioneer-dsl-2018.github.io/class/DSLs_Internal.pdf*

[4] Fowler, Martin, and Rebecca Parsons. Domain-Specific Languages. Addison-Wesley, 2011.

[5] John. "Context-free." GitHub. Accessed June 20, 2018. *https://github.com/MtnViewJohn/context-free/wiki.*

3 Language design details

In this work, a user will type in their commands in the Terminal (Mac OS) to make the drawing. In particular, they would type down each step with necessary details in a considerably general-speaking grammar. The basic computation geometry-draw performs is drawing using the StdDraw library. The overall structure table is listed below. The input contains two parts: feature points and variable. The example feature points can be "draw with ruler," "remove," or "set point." And the variable can be strings or numbers. In the program, two corresponded parts are designed to fulfill the task: identifier and evaluator. The identifier part, implemented by parser, will identify the feature points within the command line given by users and separate those feature points with the variables. Then the evaluator, implemented in the back end, will take those variables as input for certain functions and return the calculated output. In this case, the program will return certain changes in the StdDraw canvas.

Table 1. overall structure

Kind	**Detail**
Nouns	line, point, arc, circle, center, length, step
Verbs	draw, mark, remove, set, move
Adjectives	angle, color, thickness
Adverbs	counterclockwise/clockwise

The programs can go wrong in different ways such as wrong grammar, impossible definition and false structure. In designing the language, all of the above error types were carefully taken into consideration. For the wrong grammar error, the output in the Terminal will contain the error code with specified explanation. For the impossible definition, such as set a character which has not been implemented, the output will print out the possible inputs in each specific scenario. As for the false structure, like the grammar error, the output will tell the user what is being called and what is expected. This work does not have a complete error-checking tool due to the limited time. However, for the listed error solutions, a detailed explanation along with the possible remedies will be given.

Currently there are plenty DSLs for this domain in the market. Matlab and Walframe Alpha are two of the most influencial DSLs for this domain. In comparison, Matlab and Walframe Alpha concern bigger problems such as economic and scientific models while geometry-draw has a better user-friendly grammar design and is considered more domain specific.

4 Example programs

4.1 Draw the equilateral triangle

Train of thought:

1. Draw line1.
2. Mark each end point1 and point2
3. set compasses' center on point1 and its length equals the line.
4. Draw Arc1: 70 degrees angle clockwise.
5. Repeat step3 on point2.
6. Draw Arc2: repeat step4 counterclockwise.
7. Mark the intersection points between Arc1 and Arc2 point3
8. Connect point3 and point1, point2

Change the thinking into geometry-draw language:

```
> draw line with ruler from (0.5,0.5) to
(0.7,0.5)
> draw arc with compass counter-clockwise
from (0.5,0.5) with radius 0.2 from 0
to 70
> draw arc with compass counter-clockwise
from (0.7,0.5) with radius 0.2 from 110
to 180
> set point (0.6, 0.673) color to red
> draw line with ruler from (0.5,0.5) to
(0.6, 0.673)
> draw line with ruler from (0.6, 0.673)
to (0.7,0.5)
```

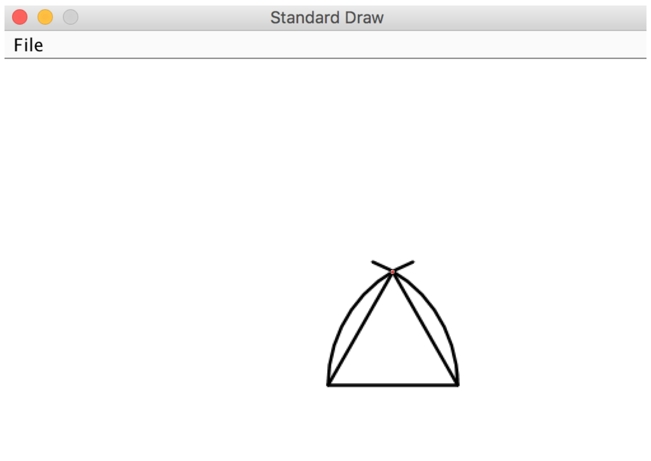

Fig. 1. Draw the equilateral triangle

4.2 Draw Angle Bisector

Train of thought:

1. Mark each side line1 and line2
2. Mark the intersection points between line1 and line2 P1
3. Set compasses' center on P1 and its length equals 5 cm.
4. Draw Arc1: 60 degrees angle counterclockwise.
5. Mark the intersection points between Arc1 and line1, line2 Point2,3
6. Set compasses' center on point2 and its length equals 3 cm.
7. Draw Arc2: 60 degrees angle counterclockwise.
8. Repeat step6 on point3
9. Draw Arc3: repeat step7 clockwise
10. Mark the intersection points between Arc2 and Arc3 point4
11. Connect point4 and P1

Change the thinking into geometry-draw language:

```
//default lines
> draw line with ruler from (0.1,0.3)
to (0.7,0.3)
> draw line with ruler from (0.1,0.3)
to (0.6,0.803)
//draw
> draw arc with compass
counter-clockwise from (0.1,0.3) with
radius 0.3 from 0 to 50
> draw arc with compass clockwise
from (0.31, 0.51) with radius 0.5
from 30 to 120
> draw line with ruler from (0.1,0.3)
to (0.8,0.6)
```

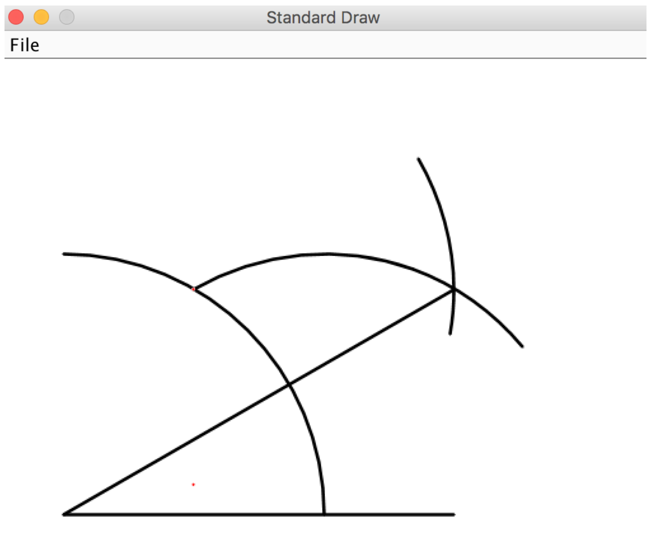

Fig. 2. Draw Angle Bisector

4.3 Draw 30 Degree Angle

Train of thought:

1. Set compasses' center on point1 and its length equals 5 cm.
2. Draw Arc1: 90 degrees angle counterclockwise.
3. Repeat step1 on point2
4. Draw Arc2: 70 degrees angle clockwise.
5. Mark the intersection points between Arc1 and Arc2 point3
6. Repeat step1 on point3
7. Draw Arc3: 30 degrees angle counterclockwise.
8. Mark the intersection points between Arc2 and Arc3 point4
9. Connect point4 and point1

Change the thinking into geometry-draw language:

```
> draw line with ruler from (0.1,0.3) to
(0.7,0.3)
> draw arc with compass counter-clockwise
from (0.1,0.3) with radius 0.4
from 0 to 90
> draw arc with compass counter-clockwise
from (0.5,0.3) with radius 0.4
from 40 to 120
> set point (0.3, 0.646) color to red
> draw arc with compass counter-clockwise
from (0.3, 0.646) with radius 0.4
from 0 to 30
> draw arc with compass clockwise from
(0.3, 0.646) with radius 0.4 from 0 to 10
> draw line with ruler from (0.1,0.3) to
(0.7, 0.646)
```

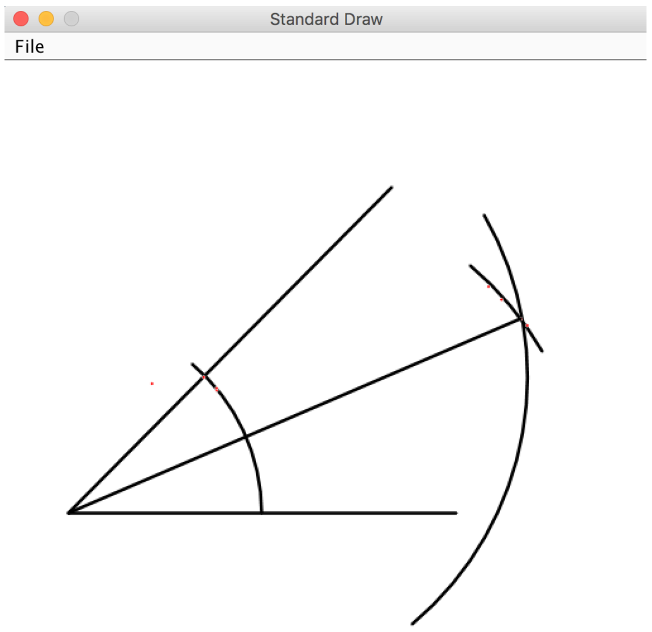

Fig. 3. Draw 30 Degree Angle

5 Language implementation

This work is considered as an external DSL. All the implementation process is done in Scala. The purpose for this DSL is to shorten the distance between coding and the public. It was designed for public to draw the geometry figure through a daily-use language. As a result, the General-Purpose Language does not provide that convenience while only an external DSL could solve this problem. The architecture of geometry-draw has three parts: the front, middle and back-end. The front end, which is also the parsing end, is the most complicated end.[6] As Grammar Design in Figure1.2 shows, the parser would first identify the action from the command given by users. After identifying the general action type, the parser will subdivide a more specific action from one of the five major action. The parser will then identify and store the variables and place them as inputs to the middle end. The parser is implemented from the JavaTokenParser and PackratParser. In particular, the PackratParser identifies the special types in the middle end such as Point, Action, and line. The general Parser identifies the string value within a given range. The most special one is the string-identifier, using the techniques of ASCII codes, which returns all the string values for the mark function. The middle end, which is also called the intermediate representation, is the transfer station between the front end and the back end.[7] In the middle end, one sealed abstract class Action is defined. Extended the Action class, 11 case class of actions are designed along with 3 case class of data types: Point, Line and Arc. Consider the specialty of the arc, case class Rotation and case class Direction are made. The major data structure case class in Scala is used due to the fact of its convenience in pattern matching and its immutability. The back end, also treated as execution part, is where program takes in the functions and variables and does the actual calculation.[8] In the back end, a Scala's unique function is used: pattern match. Due to the fact that pattern match would only runs the last line of the code if multiple lines are available, I made each pattern a correspond _do" function to avoid this problem. In the _do" function for each specific action, the StdDraw library is used. The back end also contains many mathematical calculations for angles, coordinates and reality factor. In respond to rotate direction, all the angle calculation is modified to fulfill the situation. For the reality factor, add a slightly larger pen thickness can avoid the default thickness of the frame of line when manipulating the line. Overall, compared to the semantics of Scala, the semantics of geometry-draw is much simpler. For the draw function, this DSL saves 7 lines of codes. For the move function, it even saves about 30 lines of codes. The back end not only shorten the code user needs to write, but also provides the different outputs with different situations.

[6] All the front end is implemented in *Parser.scala*.
[7] All the middle end is implemented in *AST.scala*.
[8] All the back end is implemented in *interpreter.scala*.

6 Evaluation

This work is basically on the opposite side of the General-Purpose Language. That to say, all the grammar in this language do not have any similar duplications in the General-Purpose Language. Geometry-draw works well in many areas. First, the grammar is well designed to make user achieve their goal in daily expressions. Second, the output window gives user a detailed ocular feedback that paper could not do. Third, the functions implemented have already covered a wild are of tasks within this domain as the three example programs show. In particular, I am pleased with its simple linguistic descriptive grammar. For most simple tasks required at primary school, this language has a great potential. Though, currently for some harder tasks like drawing a Pentagon, this language reaches its bottleneck due to its limited features. Therefore, Geometry-draw has various fields to improve. One thing that troubles most users is the storage system. Once the "mark" function is fully functioned, users can use nouns to replace the complicated Point or Line, which would save a lot of time as well. Besides the mark function, the auto-calculator function is another feature that I would like to implement. It can make this work capable of doing more tasks such as finding the area of an enclosed space and calculating the intersection points between two lines. Currently, this work can be seen as the most foundational version of the language. It covers most of the necessary functions for this domain, and it is the foundation for a more perfect, all-round product.

In the evaluation process, I first did many example constructions through IntelliJ to see if everything works perfectly. In this step, the main focus would be on the language itself: the fluency, host flavor and respond quality. Then I send the prototype to my friends who are the graduate students in Computer Science in Zhejiang University and ask for some feedbacks. From these evaluation processes, I get many insightful suggestions and errors I haven't thought about. I changed my grammar design based on the feedback from my professor, group mates, and my friends. Based on the result tested on MathisFun website, I found many useful features I could add to my language. Many significant changes are made such as the remove function and the characteristics of the data type Point and Line.

Throughout the implementation process, I ran into trouble several times due to the lack of experience in Scala. The first major problem I faced is the case match function. It would only return one output while the expected outputs are three. After reading lots of materials and watching several YouTube videos, I eventually get to have a _do function to solve the problem. The second major problem is the parser. I ran into trouble with Parser for more than two weeks due to the limited knowledge. In this process, the professor made great help: the external-lab and suggestions on LMS. The third major problem is the "mark" function that is still left unfinished. The problem is the case class is immutable, so when I try to reassign a string value in the case class, the program gives the error code and failed to store the Point data type into the string.