Bilkent University

CS315 - Programming Languages Course Project

P.S!

A Letter-Format Drawing Language

Nashiha Ahmed	21402950
Mert Inan	21402020
Cholpon Mambetova	21402612

Instructor: Dr. Buğra Gedik

Section: 2

Date: 28th Nov 2016

Contents

1 | Introduction

The main ideology behind P.S! is describing a shape to a friend as if writing a letter. The name P.S! comes from the P.S. (Post Scriptum) in a letter, and the ! denotes a special meaning in our program, as explained in later sections. P.S! is a letter type programming language and will be compiled by a compiler named *Kukubo*, which means "drawing robot" in Japanese. The source code will look like an actual letter that people write and send to a friend giving instructions on some tasks. Therefore, the language is as human readable as possible, which is what makes it different from most other programming languages. Since P.S! is a figure-drawing programming language the main purpose of it is to draw shapes, from basic built-in ones to complex ones defined by a user. The programming language is made in such way that a user needs to be polite for the program to work. There are some reserved words like "please", "thank you" or "dear", that make sure that a user is polite in that sense. Therefore, we can say that authors of P.S! are propagating humanness and politeness. The idea of creating a letter type programming language came to us, while we were brainstorming for this CS315 project. While brainstorming we started to imagine that we were giving instructions to a friend. This is when we had our "eureka!" moment.

2 | The Entry Point

From the standpoint of high-level programming languages, a main entry point is the first subprogram that needs to be executed. P.S! gives utmost priority to the main function and makes it the key component of the language. As P.S! has a letter format, body of the letter represents the main function. Some high-level languages choose not to include the main function as the starting point, yet as user-defined functions will be used less than the main drawing functionalities, it is easier and effective to give the main function a bigger area in the overall structure of a letter of code. The reserved words to define a main function are `Dear Kakubo,` and `Best wishes.` The code between these reserved words are considered to be in the main method and will get executed first. In order to increase writability, capitalization of the reserved words are ignored. Hence `dear kakubo` and `best wishes` are also accepted. User-defined functions and shapes are written after the main function using `P.S.` and `P.P.S` reserved words, respectively. The end of the whole letter and the source code is represented with an exclamation mark. Following is the general structure of a P.S! code.

```
Dear Kakubo, (Start of a main)
________________________________________________
________________________________________________
________________________________________________
________________________________________________
Best Wishes. (End of a main)

P.S. (User-defined function definitions)
________________________________________________
________________________________________________

P.P.S. (User-defined shape definitions)
________________________________________________
________________________________________________
! (End of a file)
```

If the source code doesn't have a `P.S.` or `P.P.S.` parts, then the `!` will end the program right after the `Best Wishes`. For commenting the parentheses are used.

3 | Drawing the Basics

Drawing Statements

Since it is a figure drawing language, it is essential that the drawing keys are user friendly and flexible in use. The most important built-in function of this programming language is the `draw` function, and the easiest form of the drawing statement would look as the following:

```
Draw an oval.
```

The compiler will determine the reserved words `draw` and `oval` and will stop reading the parameters when it sees the dot. The `draw` is the built-in function in P.S! that draws a shape, and the `oval` is a built-in shape. Since there are no parameters in our example, the program will draw an oval using default parameters (location, size, stroke width, color, fill state and fill color).

A more complex statement might look as the following:

```
Draw an oval at position [15, 56] with height [4] using stroke
[2.5] also color blue also name "My-Circle".
```

The order of parameters does not matter, only the reserved words as `position`, `radius`, `stroke` (stroke width), `color`, `name`, etc. followed by specified types, do matter. The `position` should be followed by two integers in square brackets separated by a comma, `stroke` with a float, `name` by a string, `color` by reserved word for color or RGB code, etc. Parameters can be separated by words `using`, `at`, `with` and `also`. The program reads parameters, until it sees a dot which is the end of a statement. The parameters that were not specified by a user will have a default value, except for the name. If the figure was not named, it will remain unnamed and cannot be referenced later. Since, we can omit all the parameters, they are all optional, which makes the language flexible.

The drawing strings are also can be drawn using the following statement:

```
Draw a String "P.S! is the best language. It supports letter-type
languages that are nice and polite and lead users to be nice and
polite. Therefore, P.S! leads to peace in the world!" at position
[58] and [104].
```

It might have parameters as `position`, `color` (color of letters), `stroke` (stroke width), `size` (font size), `border state`, `border color`, `fill state`, `fill color`. The `text` parameter is mandatory, all other ones are optional, so if not specified, the default values will be used.

Scale & Stroke

Some of the most important parameterizable variables of the main draw function are the scale and stroke. The draw function can be both used with and without parameters of scale and stroke. Most important parameters are width and height for scale, and stroke itself. These parameters are recognized using the `height`, `width` and `stroke` reserved words. If the user chooses to not use the parameters, then default values for scale and stroke are used to draw. For instance, in order to draw 10 ovals next to each other iteratively without using scale parameter, the following code can be executed which uses the iterative `times` loop.

```
(Assuming that a value for "count" variable is set at P.P.S.)
Please,
Draw an oval at position [count, 100].
count is count plus oval's width.
Thank you, [10] times.
```

In this example, in order to refer to the drawn oval's width argument, the keyword oval is used even though its specific width is not explicitly written. The `times` loop will be discussed in detail in section 5.

Location, Size & Color

Since location, size, and color are essential in a figure-drawing language, our language P.S! supports these types. The usage of these types are simple. The location type has two members x and y. For example, if the user wants to draw a shape at a specific position, the user simply needs to type: `position[`x-value, y-value`]`. The default position is `[0,0]`.

```
Draw an oval at position[10,10].
```

The size type also has two members width and height. If the user wants to draw a shape with specific width and height, the user need only type: `width[`desired-number`]`or `height[`desired-number`]`. The specified width and height are actually the width and height of the bounding box of the shape.

```
Draw an Oval using width [15] also height [10].
```

In the above example, the bounding box within which an oval is drawn has width of 15 units and height of 10 units.

Lastly, P.S! has predefined colors with their specific RGB values. These colors are red, blue, green, yellow, white, and black. The user may also draw with colors not built in by specifying their RGB values. This can simply be done by typing `color holding rgb[`desired red, blue, green values`]`. Some examples are shown below.

```
Draw a rectangle using color blue.
Draw a rectangle using color holding rgb[ 204,204,255].
(Which is periwinkle blue)
```

Shapes

The user can draw shapes from two categories: built-in shapes and user-defined shapes. Shapes have optional parameters. Both shapes, unless parameters are specified, are drawn with default parameters and within a default bounding box. Further descriptions of the two categories are explained in the two subsections. The shapes are scale free in this way.

Built-In

There are a few shapes built-in the language. These are: line, rectangle, and oval. These built-in shapes can be drawn with optional parameters. If essential parameters are not used, then the shape will be drawn with default specification, and in a default bounding box. The default bounding box has height and width of 5 units. If the user does not specify the position at which a shape will be drawn, the shape will be drawn at the origin (0,0). Lines can be drawn in 8 directions: north, north-west, west, south-west, south, south-east, east, and north-east. Lines can have arrows on both ends of a line, and arrow sizes are relative to the stroke width. An Oval does not specify any parameters. An oval's size is determined by its bounding box. A rectangle can specify a parameter about having its corners rounded. Some example code are as follows:

```
Draw a Line using direction north also start arrow also
stroke-width [1].
```

```
Draw a Rectangle using rounded-corners also width [10].
Draw an Oval using length [20] at position[10,10].
```

User-defined

The user can define new shapes that are a composition of the built-in shapes. These shapes are explained in detail under the extension of shapes section. These shapes are also parameterizable and can be used once they are defined in a `P.P.S` module.

4 | Dynamic Typing

P.S! supports primitive types supported in other programming languages as `float`, `integer`, `string`, `char` and `boolean`. Numbers as float and integers are written in square brackets, as `[5]` or `[-3.4]`. Strings are written in double quotation marks, like `“Some text”` and chars are written in single quotation marks as `‘a’`. Booleans can be written as `true` or `false`, as well as `1` and `0`.

The basic arithmetic and logical operations are supported as well. Those operations have a few versions user can write them.

- Assigning can be written as `assign` or `is` , besides regular = sign.
- Addition can be written as `add`, or `plus`, besides regular + sign.
- Subtraction can be written as `subtract` or `minus`, besides – sign.
- Multiplication can be written as `multiply`, besides `*` sign. (Be careful, `times` cannot be used here)
- Division can be written as `divide`, besides / sign.
- Modulo can be written as `modulo` or `remainder of`, besides `%` sign.
- Logical AND can be written as `and`, besides `&&` sign.
- Logical OR can be written as `or`, besides `||` sign.
- Logical NOT can be written as `is not`, besides `!=` sign.

The strings can be concatenated when the addition operator is used. For example,

```
My-String1 is “Letters”.
My-String2 is “cool”.
My-Final-String is My-String1 + “ are ” + My-String2 + “!”.
```

As the result, `My-Final-String` will be “`Letters are cool`!” by concatenation.

String characters can be converted from integers using ASCII code, and the other way around. The following is the example of such conversion:

```
My-Char is ~char~ 65.
My-Int is ~int~ ‘w’.
```

5 | Loops

In most programming languages, looping is done using conditions and increments. In order to incorporate loops fully, P.S! makes use of two kinds of loops: iterative and conditioned. Iterative loops mainly repeats tasks for a certain amount of times. This loop is very similar to "for" loops in Python and Java, yet it only accepts numbers as variables. This design decision was made, as drawing requires repetitive tasks which can be represented just with numbers. Hence providing this kind of a loop helps the user to write repetitive tasks relatively easily. The reserved word for iterative loop is `times`. In order to use the loop, a comma followed by a number and `times` keyword should be used. An example use is given in the sample code below.

```
Draw a circle, [2] times.
```

Conditioned loops are the second type of loops that are provided by P.S!. They are very similar to "while" loops in other programming languages. This loop can be considered as an extension for the iterative loop. As an addition to the iteration variable, it has a condition statement which is checked after each execution of the loop. `until` is used before the condition to denote the conditioned loop. An example code is as follows.

```
Until, position is less than or equal to [5, 3], draw a circle.
```

In this example, the body of the loop is the `draw a circle` statement. If the body takes more than one statement that cannot be ended with a dot, then `Please` and `Thank you` keywords are required to create a block. Both loops can be combined as in the following example.

```
Until, NUMBER is greater than [5],
Please, draw a oval, [2] times. NUMBER is NUMBER plus 5. Thank you.
```

6 | Conditionals

P.S! supports conditional statements using the regular `if` and `else` reserved words. As conditional statements are a crucial part of a programming language, it was better to use a familiar wording. However, in order to provide flexibility to the user while writing, synonyms of `if` and `else` can be used as well, such as: `with the condition`, `in case`, `on the occasion that`; `if not`. `Else` keyword can be used with and without a comma at the end. The conditional statement is used by writing the reserved word followed by the condition. Every condition statement has to be preceded and followed by a comma. A conditional block is given in the following code sample.

```
With the condition that, Face-Number is smaller than [5], draw a
Face, at position [100, 200]. If not, Face-Number is Face-Number
plus [1].
```

If the conditional body requires more than a single statement, then it is needed to group the statements using `Please` and `Thank you` keywords, as usual. An example use can be as follows.

```
Until, House-Number is not [0],
Please,
If, House-Number is greater than or equal to [100],
Please, draw Congrats-String. Draw a House. Draw a Horse. Draw a
Person. Thank you.
Thank you.
```

7 | Functions

The P.S! supports user-defined functions. The user can define a function at the `P.S.` section, that is after the entry point that ends with `Best Wishes`. The followings are the examples of the declaration of user-defined function:

```
P.S. (The user-defined functions section start)

(Definition of Function 1)
I defined the function FILL-WITH-STRIPS as the following:

Please,
 ____________________________________________________________________
 ____________________________________________________________________
Thank you.

(Definition of Function 2)
I defined the function DRAW-FACE with ID as the following:

Please,
 ____________________________________________________________________
 ____________________________________________________________________
Give back ID.
Thank you.
```

Every block after `P.S.` that starts with `I defined the function + ID + as the following: please` and ends with `Thank you.` means the function definition. The user can describe a function between `please` and `Thank you`. If the function uses some parameter, then those parameters need to be written by `with` keyword followed by parameters. If the function returns some variables, then it should end with `return` or `give back` and a variable to be returned.

8 | Extension of Shapes

Our language also supports the extension of shapes. That is, the user may define new shapes based on the shapes that are already built into the language. These shapes are parameterizable. They may include optional or mandatory parameters as the user pleases. If essential parameters are not specified, default ones defined by the language will be used. To define a shape, the user can start a new module, starting with `P.P.S. I defined` then add the name of the shape and continue with `as the following:` If the user wants to add parameters, the user can do as follows: `P.P.S I defined` [user's shape] `with` [the desired parameters] `as the following:` Once the user has finished the module ending it with `Thank you.` the user can now draw this shape. An example of the extension of shapes is shown below:

```
P.P.S. (The user-defined shape section start)

(Definition of Shape 1)
I defined Triangle with name Happy-Triangle as the following:

Please,
Triangle has 3 lines.
______________________________________________________________
______________________________________________________________
Thank you.

(Definition of Shape 2)
I defined Circle with number and position-X as the following:

Please,
Circle is an Oval.
Circle’s width is equal to Circle’s height.
______________________________________________________________
______________________________________________________________
Thank you.
```