The Zig Language Specification

Geoffrey Salmon & Karmen Liang

Introduction

Production of computer graphics is a creative process; programs like Photoshop, Gimp, and, yes, even Microsoft Paint, allow the user's creative energy to run free. When it comes to a more algorithmic, geometric approach, however, humans cannot compete with computers. Programming languages like Logo serve this purpose and can be an educational way to learn geometry and programming for newbies. However, its educational bent, while valuable, provides an interface that is not optimal.

Zig strives to emphasize syntactic and semantic transparency and readability in order to create a clean, consistent, and powerful interface for the production of vector graphics. It utilizes an intuitive syntax with simple features like loops and variable assignment. By manipulating the primitive constructs of the turtle and the pen, users can create SVG graphics that range from squares to gradient spirals.

Design Principles

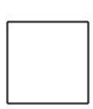
In Zig, no meaning or functionality is hidden in class hierarchy. While emulating natural languages (as Logo does), clarity of logical flow takes the forefront. Semicolons and brackets provide basic structuring in delineating code, but whitespace is what separates most primitive functions from their arguments. This aids in readability and ease of use.

Examples

```
// primitives
ahead 50        // moves the turtle ahead by 50 steps
clockwise 45    // rotates the turtle 45 degrees clockwise
```

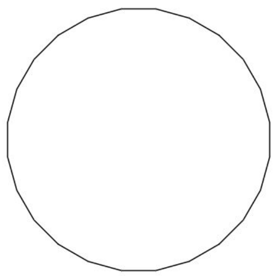
1.

```
// draws a square with side lengths of 50 pixels

dotnet run "ahead 50; clockwise 90; ahead 50; clockwise 90;
ahead 50; clockwise 90; ahead 50"

output:
```



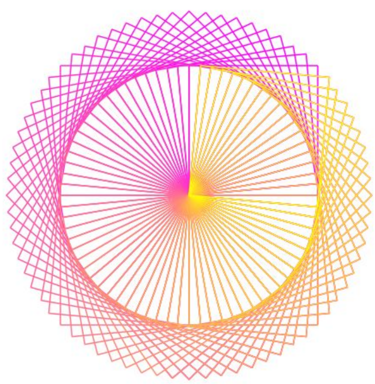
2.

```
// draws a (rough) circle using the looping construct

dotnet run "loop (24){ahead 30; clockwise 15}"

output:
```



3.

```
// draws a gradient spiral
dotnet run "let x = 0; let y = 255; loop (100){clockwise 5; penrgb 255,y,x;
loop (4){ahead 100; clockwise 90}; x += 4; y -= 4}"

output:
```



Language Concepts

The programmer must understand the concept of the turtle (the item which is moved around) and the pen (the item that draws the line. The basic commands all follow from these two ideas. If the programmer understands how to build modularly, they will succeed. Primitives of the language also aid in compositionality; these include the basic commands and the numbers given as arguments to these commands.

Syntax

Basic commands:

```
COMMAND           SHORT FORM    DESCRIPTION
ahead x           a x         // move turtle ahead by x steps
behind x          b x         // move turtle back by x steps
clockwise x       cw x        // rotate turtle x degrees clockwise
counterwise x     ccw x       // rotate turtle x degrees counterclockwise
home                          // move turtle to center screen
lift                          // move turtle without drawing
press                         // resume drawing
clear                         // clear the screen, return turtle home
pencolor x        pc x        // set pen color
penrgb r,g,b                  // set pen color according to r,g,b
penred x                      // set red component of RGB to x
penblue x                     // set blue component of RGB to x
pengreen x                    // set green component of RGB to x
```

Control flow:

```
loop (x){expression}          // executes expression x times
```

Formal Syntax:

```
<expr>        ::= <seq>
               |  <nonrecexpr>
<nonrecexpr>::= <command> <input>
               |  <command> <input>,<ws><input>
               |  penrgb <input>,<ws><input>,<ws><input>
               |  <command>
               |  <var>++
               |  <var>--
               |  <var><ws>+=<ws><number>
               |  <var><ws>-=<ws><number>
               |  let <var><ws>=<ws><input>
               |  <loop>
<seq>         ::= <nonrecexpr>; <expr>
<command>     ::= ahead       | a
               |  behind      | b
               |  clockwise   | cw
               |  counterwise | ccw
```

```
                |  press
                |  lift
                |  pencolor    | pc
                |  penred
                |  pengreen
                |  penblue
                |  penwidth    | pw
                |  home
                |  sethome
                |  setdim
                |  pen
<input>       ::= <number> | <string> | <var>
<loop>        ::= loop (<number>)<ws>{<nl><ws><expr><nl>}
<varlist>     ::= <var>, <varlist>
                |  <var>
<var>         ::= <letter><var>
                |  <letter>
<letter>      ::= a | b | … | z
                |  A | B | … | Z
<character>   ::= <letter>
                |  <number>
<number>      ::= <d><number>
                |  <d>
<d>           ::= 0 | 1 | … | 9
<ws>          ::= " "<ws>
                |  " "
                |  ""
<nl>          ::= "\n" | "\n"<nl>
```

Semantics

Syntax	Abstract Syntax	Meaning
`ahead` *n* `a` *n*	`Ahead of Expr`	Moves turtle forward by *n* pixels
`behind` *n* `b` *n*	`Behind of Expr`	Moves turtle backward by *n* pixels
`clockwise` *n* `cw` *n*	`Clockwise of Expr`	Turns turtle clockwise by *n* degrees

`counterwise` *n* `ccw` *n*	`Counterwise of Expr`	Turns turtle counterclockwise by *n* degrees
`press / lift`	`Press / Lift`	Press places pen down for drawing. Lift allows turtle to be moved without drawing.
`ahead` *n*`; clockwise` *n*	`Seq of Expr*Expr`	Evaluates the first expression followed by the second expression
`pencolor red` `pc red`	`Pencolor of string`	Changes the pen color to be that of the string given
`penred` *n*	`Penred of Expr`	Changes pen's red RGB value to *n*
`pengreen` *n*	`Pengreen of Expr`	Changes pen's green RGB value to *n*
`penblue` *n*	`Penblue of Expr`	Changes pen's blue RGB value to *n*
`penwidth` *n* `pw` *n*	`Penwidth of Expr`	Changes pen width to *n* pixels
`ahead 50; cw 90`	`Seq of Expr*Expr`	Executes a sequence of instructions
`loop (`*n*`){ahead 50}`	`Loop of Expr*Expr`	Repeats the expression in the braces *n* number of times
`let x = 5`	`Assign of string*Expr`	Map a value to string, adding it to the Context Map
`x++`	`UnaryIncrement of string`	Increments the variable x by 1
`x+=1`	`Increment of string*int`	Increments the variable x by the amount specified
`x--`	`UnaryDecrement of string`	Decrements the variable x by 1
`x-=1`	`Decrement of string*int`	Decrements the variable x by the amount specified
`penrgb 75,150,100`	`PenRGB of Expr*Expr*Expr`	Sets the pen color to the RGB value
`home`	`GoHome`	Returns the pen and turtle to the starting point
`sethome x,y`	`SetHome of Expr*Expr`	Sets the starting point to the coordinates x,y
`setdim x,y`	`SetDimensions of Expr*Expr`	Sets the dimensions of the canvas to x by y pixels

Primitives

Numbers, strings, and booleans comprise of the primitives of the language. Strings allow for the naming of new functions, while numbers are passed in as arguments to functions. In addition, vector lines form the basis for each command that is carried out by the virtual pen and turtle; from this we understand lines to be a primitive type. The window space that the lines are drawn in (the canvas) may also be understood as a primitive type. Booleans are used in control structures to provide logical flow.

Representation

At a high level, the program is represented with a `State` type, in which the current states of the `Canvas`, `Turtle`, and `Pen` are maintained and updated with every function. `Canvas` contains a list of `Lines` that have been drawn with the `Pen`. Turtle maintains the x and y position of the turtle item, as well as the angle it is pointing towards. `Pen` contains information about the thickness and color of the line, as well as whether or not it is being pressed down on the canvas. Since every function in this language is essentially side-effecting in respect to the `State`, each function should return a new copy of the updated `State`. The output of the program is an SVG file that is opened up in a web browser.

Remaining work

Functions

An essential feature of the language is the ability to bind functions to a function name (essentially, a function variable). In the end, we hope that users will be able to define functions such as:

```
func square length {
    loop (4) {ahead length; cw 90;}
}
```

Lexical scoping

We currently have global and dynamically-scoped variables. We hope to implement lexical scoping by passing around a Context Map that is tied to a Scope. The Scope should be nested Context Maps that creates separate environments for variables (Scope is a data type that contains a Context Map and a parent Scope).

Eraser, screencolor

Allowing the user to change the SVG background color allows the use of an eraser, which is essentially a pen whose color matches that of the background.

Constraints

The prior draft including goals that resembled those above. Functions and lexical scoping were largely the focus of the final implementation. However, due to time constraints, we decided to round out the expressive features of the language (pen colors, setting dimensions, etc) instead of pursuing the more complex implementation of functions.