COMPILER DESIGN FOR
A NEW ESOTERIC LANGUAGE

Vinayak Kaushik (15BCB0028)

INTRODUCTION

The main aim of a compiler is to convert source code in a high-level language into code in a low-level language.

Due to the complexity of compiling languages, the process is split into several phases, which helps ensure that the overall process produces high-quality output and be easily maintained. The separation of stages helps ensure compiler correctness, which is extremely important due to compiler errors being so difficult to track down.

What I propose to do is to create a combination of two Esoteric Programming languages. “Piet” and “BrainFox”. These two language’s have their own syntax’s and different characteristics.

I would be using the syntax of the Brainfox language and then convert it into a graphic image for storing the program.

PIET

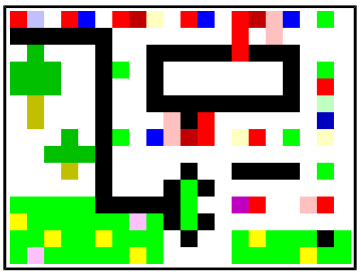
Piet is a programming language in which programs look like abstract paintings. A program in Piet differs from a program in a conventional programming language since it is a colourful picture.

light red	light yellow	light green	light cyan	light blue	light magenta
red	yellow	green	cyan	blue	magenta
dark red	dark yellow	dark green	dark cyan	dark blue	dark magenta
white			black		

	darkness change		
Hue change	**None**	**1 Darker**	**2 Darker**
None		push	pop
1 Step	add	subtract	multiply
2 Steps	divide	mod	not
3 Steps	greater	pointer	switch
4 Steps	duplicate	roll	in(number)
5 Steps	in(char)	out(number)	out(char)

In a Piet program the colours themselves do not matter, it's the **transitions** in hue and darkness that form the code.

Example Program:



BRAINFOX

BrainFox is represented by an array with 30,000 cells initialized to zero and a data pointer pointing at the current cell.

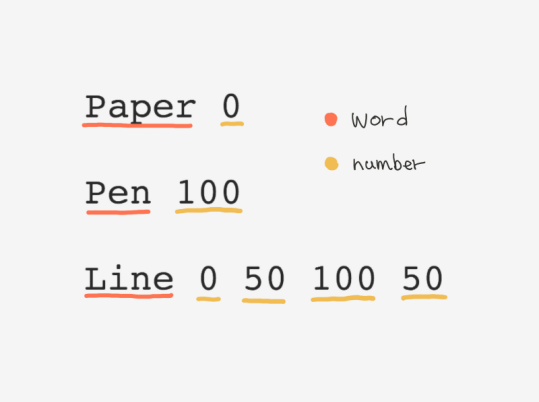
There are eight commands:
+ : Increments the value at the current cell by one.
- : Decrements the value at the current cell by one.
> : Moves the data pointer to the next cell (cell on the right).
< : Moves the data pointer to the previous cell (cell on the left).
. : Prints the ASCII value at the current cell (i.e. 65 = 'A').
, : Reads a single input character into the current cell.
[: If the value at the current cell is zero, skips to the corresponding] .
Otherwise, move to the next instruction.
] : If the value at the current cell is zero, move to the next instruction.
Otherwise, move backwards in the instructions to the corresponding [.

[and] form a while loop. Obviously, they must be balanced.

COMPILER DESIGN

1. Lexical Analysis (tokenization)

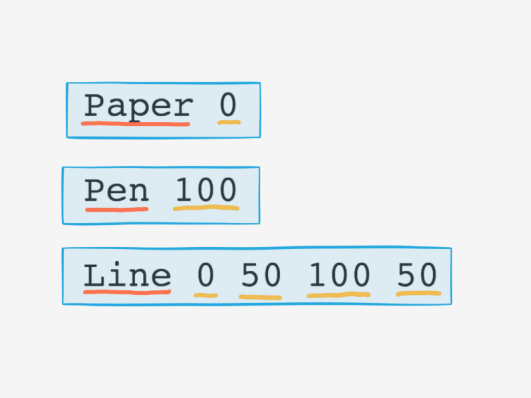
First thing we did was to separate each keywords (called tokens) by white space. While we are separating words, we also assigned primitive types to each tokens, like “word” or “number”.



2. Parsing (Syntactical Analysis)

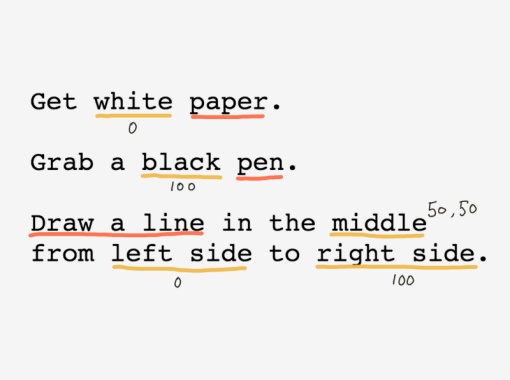
Once a blob of text is separated into tokens, we went through each of them and tried to find a relationship between tokens.

In this case, we group together numbers associated with command keyword. By doing this, we start seeing a structure of the code.



3. Transformation

Once we analyzed syntax by parsing, we transformed the structure to something suitable for the final result. In this case, we are going to draw an image, so we are going to transform it to step by step instruction for humans.



4. Code Generation

Lastly, we make a compiled result, a drawing. At this point, we just follow the instructions we made in previous step to draw.

PROPOSED IMPLEMENTATION

I will be writing a compiler for this new language in “Python 3.5”

I will be making use of the ‘PILLOW library’ to manipulate graphics and store program’s.