Crayon Language Design

The crayon interpreter is a very basic, not overly useful execution of instructions written in the "Crayon" toy programming language. It did intend to explain language functions better although changing the keywords to colours actually appears to increases the difficulty. The interpreter is written in python and uses logical computation with an imperative structure. Although grammar is quite different to most programming languages, it's syntax rules borrow heavily from python. To give a concise insight of crayon through brief code samples and text, the following sections will cover classes and operation, achieved functionality, attempted functionality and classes and operation.

Achieved Functionality

Crayon Grammar Function Map

Crayon:	Function:
RED	Var
BLACK	String
BLUE	Multiple Statement
PURPLE	If
GREEN	Then
YELLOW	Else
VIOLET	While
ORANGE	Do
TAN	End
APRICOT	And
FERN	Or
WHITE	Not
PEAR	For
BROWN	Print

Print

Very limited, only statement results are printed to the console ie: mathematical outcomes, or If statements. (Displayed using all test files)

Relational and Boolean

Operators exist as normal to limit confusion

```
['<', '<=', '>', '>=', '=', '!=']
```

Boolean expressions: And, or, not

```
        var1 RED 2 BLUE
        var2 RED 3
Var1 < 3 APRICOT var2 < 4 = True
Var1 < 3 FERN var2 > 4 = True
WHITE  var2 < 4 = False
```

Var, Strings and Multiple Assignments

Numbers and Strings can be assigned to variables using a particular id name. (Displayed in var.cryn test file)

```
var1 RED 2 BLUE
```

Var1 is variable ID, RED assigns as a variable and BLUE represents multiple statements similarly to a semi-colon. Required when continuing statements. Strings work in the same way.

```
string1 BLACK HelloWorld BLUE
```

Although spaces are not allowed, nor string manipulation. (Displayed in string.cryn test file)

If Statements and While Loops

If statements work using If, Then, else and end keywords using indentation: Statements also work with strings. (Displayed using if.cryn test file)

```
PURPLE x != 2 GREEN
        y RED 2
YELLOW
         y BLACK IfString
                  TAN
```

While Loops work using While, Do, and End keywords: While statements do not work with strings do to lack of print function. (Displayed using while.cryn test file)

```
VIOLET x < 10 ORANGE
        x RED x + 1
                  TAN
```

Both statements can also be nested (Displayed using nested.cryn test file)

Mathematic Expressions

Expression calculations can be applied to numbers with variable result printing to console(Displayed using var.cryn test file)

```
var1 RED 2 + 2 * 4 BLUE
```

Precedence is applied using:

```
expressionPrecedence = [

   ['*', '/'],
   ['+', '-'],
]
```

Data Structure

Very limited, only basic data structure implemented is the list that holds variables using multiple assignments.

Attempted Functionality

Attempted to make floating point numbers, managed to lex them using

```
(r'\d+[eE][-+]?\d+|(\.\d+|\d+\.\d+)([
eE][-+]?\d+)?', FLOAT),
```

Unfortunately on testing they kept causing a parse error. A proper print function using BROWN, was also attempted by accessing the token stream in a similar way to Crayon.py that produces a result, this interfered with the overall operation of the test.

Crayon Classes and Operation

CrayonGrammar.py

Language markers

```
KEYWORD = 'KEYWORD'
INT      = 'INT'
CHARS   = 'CHARS'
FLOAT   = 'FLOAT'
PRINT   = 'PRINT'
```

Expression tokens

```
# Marker value map
(r'[0-9]+',                 INT),
(r'[A-Za-z][A-Za-z0-9_]*', CHARS),

# Expression operators
(r'\+', KEYWORD),
(r'-',  KEYWORD),

# grammar examples
(r'RED',    KEYWORD), # Var
(r'BLUE',   KEYWORD), # Multiple vars
(r'PURPLE', KEYWORD), # if
(r'GREEN',  KEYWORD), # then
```

CrayonSyntaxTree.py

Each syntax tree function maps to a CrayonParser.py function The syntax tree offers classes that structure the input of tokens before parsing.

```
# Assign all statements (INTS, STRINGS…
etc) against name and expressions then
return value
class AssignStatement(Statement):
   def __init__(self, name, expression):
```

In the AST: assigning strings, multiple statements, if statements, while loops, work in similar way. Mathematical, Binary, Relational and boolean expressions are also mapped to the AST using operational classes.

Crayon.py

Main method opens testfile, lexes tokens, assigns them to the AST or outputs parse error if tokens are illegal. Then prints the resulting crayon variables.

CrayonLexer.py

Attempts to match CrayonGrammar.py expression tokens to lexed chars using regular expressions

```
# Append the text and marker (Keyword)
# To the token stream
if match:
   text = match.group(0)
      if marker:
          token = (text, marker)
          tokens.append(token)
      Break
```

Error checking in place to prevent symbols not found in language markers from entering the token stream

```
if not match:
           sys.stderr.write('Unaccepted
crayon char: %s\\n' % chars[pos])
```

CrayonParser.py

CrayonParser.py holds the parsing functions that process information coming from CrayonSyntaxTree.py.

```
# Keyword parser
def keyword(kw):
   return Keyword(kw, KEYWORD)

# Token values are converted into python
values
num = Marker(INT) ^ (lambda i: int(i))
str = Marker(CHARS)
```

Statements are assigned, processed and parsed using keywords then return an AST function value.

```
def statementAssign():
   def process(parsed):
       ((name, _), exp) = parsed
       return AssignStatement(name, exp)
   return id + keyword('RED') +
expression() ^ process
```

CrayonParserSubClasses.py

Contains sub classes required for parsing and AST construction following are class headings and operation

```
# Output on successful parse
class ParseOutput:
   # Output produces AST value and stack
position argument
   def __init__(self, value, pos):

# Define parser object to take a stream of
input tokens
class Parser:
   # Method takes lexer tokens and index
   def __call__(self, tokens, pos):

# Parse keywords
class Keyword(Parser):
   def __init__(self, value, marker):

# Match any token with KEYWORD, INT,
STRING, FLOAT, PRINT
class Marker(Parser):
   def __init__(self, marker):

# Links left and right parser input to
output a value for manipulation
class Link(Parser):
   def __init__(self, left, right):

# Generate or match lists for multiple
assignments
class List(Parser):
   def __init__(self, parser):

# Allows manipulation of output values,
used to build abstract syntax tree
class Process(Parser):
   def __init__(self, parser, function):
```

An expression match parser exists for comparing separated statements and assignments, processing the next one in the queue and outputting them.

With sub-classes in place the AST can now be constructed using CrayonSyntaxTree.py

Programming Tutorials Bibliography

1. Wareham, R. (2018). *Creating a toy language with the Python, LLVM and the IPython web notebook, part 1.* [video] Available at: https://www.youtube.com/watch?v=G78cTmgeUxI&t=870s [Accessed 25 May 2018].
2. How Code (2014). *Make Your Own Programming Language - Part 1 - Lexer.* [video] Available at: https://www.youtube.com/watch?v=LDDRn2f9fUk [Accessed 25 May 2018].
3. Conrod, J. (2011). *A simple interpreter from scratch in Python (part 1).* [online] Jayconrod.com. Available at: https://jayconrod.com/posts/37/a-simple-interpreter-from-scratch-in-python--part-1- [Accessed 25 May 2018].