

Shork#

Miss Ylva Llywelyn

2023/10/14

Based on the tutorial series by David Callanan.

CONTENTS

Grammar.....	1	Lexer.cs	8
Code Listing.....	2	ShorkError.cs	14
NodeBase.cs	2	Token.cs	15
Parser.cs	4	TokenType.cs	16
ParseResult.cs	7		

GRAMMAR

This is a notation for writing down the grammar of the language. It uses regex syntax, with the components themselves being italicised.

<i>statements</i>	<code>NEWLINE* <i>statement</i> (NEWLINE+ <i>statement</i>)* NEWLINE*</code>
<i>statement</i>	<code>KEYWORD:RETURN <i>expression</i>? KEYWORD:CONTINUE KEYWORD:BREAK <i>expression</i></code>
<i>expression</i>	<code>KEYWORD:VAR IDENTIFIER = <i>expression</i> <i>comparision_expression</i> ((KEYWORD:AND KEYWORD:OR) <i>comparision_expression</i>)*</code>
<i>comparision_expression</i>	<code>KEYWORD:NOT <i>comparision_expression</i> <i>arithmatic_expression</i> ((== != < <= > >=) <i>arithmatic_expression</i>)*</code>
<i>arithmatic_expression</i>	<code><i>term</i> ((+ -) <i>term</i>)*</code>
<i>term</i>	<code><i>factor</i> ((* /) <i>factor</i>)*</code>
<i>factor</i>	<code>(+ -)? <i>factor</i> <i>exponent</i></code>
<i>exponent</i>	<code>call (^ <i>factor</i>)*</code>

CODE LISTING

NODEBASE.CS

Listing 1: NodeBase.cs

```
1 namespace ShorkSharp
2 {
3     public abstract class NodeBase
4     {
5         public Position startPosition { get; protected set; }
6         public Position endPosition { get; protected set; }
7
8         protected NodeBase(Position startPosition, Position endPosition)
9         {
10             this.startPosition = startPosition.Copy();
11             this.endPosition = endPosition.Copy();
12         }
13     }
14
15     public class CodeBlockNode : NodeBase
16     {
17         public List<NodeBase> statements;
18
19         public CodeBlockNode(IEnumerable<NodeBase> statements, Position
20             ↪ startPosition, Position endPosition)
21             : base(startPosition, endPosition)
22         {
23             this.statements = statements.ToList();
24         }
25
26         public override string ToString()
27         {
28             return string.Format("{{{0}}}", string.Join(", ", statements));
29         }
30
31         public class NumberNode : NodeBase
32         {
33             public Token numToken { get; protected set; }
34
35             public NumberNode(Token numToken)
36                 : base(numToken.startPosition, numToken.endPosition)
37             {
38                 this.numToken = numToken;
39             }
40
41             public override string ToString()
42             {
43                 return string.Format("({0})", numToken);
44             }
45         }
46
47         public class StringNode : NodeBase
48         {
49             public Token strToken { get; protected set; }
50
51             public StringNode(Token strToken)
52                 : base(strToken.startPosition, strToken.endPosition)
53             {
54                 this.strToken = strToken;
55             }
56         }
57     }
58 }
```



```

57     public override string ToString()
58     {
59         return string.Format("({0})", strToken);
60     }
61 }
62
63 public class ListNode : NodeBase
64 {
65     public List<NodeBase> elementNodes;
66
67     public ListNode(IEnumerable<NodeBase> elementNodes, Position
        ↪ startPosition, Position endPosition)
        : base(startPosition, endPosition)
68     {
69         this.elementNodes = elementNodes.ToList();
70     }
71
72     public override string ToString()
73     {
74         return string.Format("[{0}]", string.Join(", ", elementNodes));
75     }
76 }
77
78
79 public class VarAssignNode : NodeBase
80 {
81     public Token varNameToken { get; protected set; }
82     public NodeBase valueNode { get; protected set; }
83
84     public VarAssignNode(Token varNameToken, NodeBase valueNode)
85         : base(varNameToken.startPosition, valueNode.endPosition)
86     {
87         this.varNameToken = varNameToken;
88         this.valueNode = valueNode;
89     }
90
91     public override string ToString()
92     {
93         return string.Format("{0}={1}", varNameToken, valueNode);
94     }
95 }
96
97 public class VarAccessNode : NodeBase
98 {
99     public Token varNameToken { get; protected set; }
100
101     public VarAccessNode(Token varNameToken)
102         : base(varNameToken.startPosition, varNameToken.endPosition)
103     {
104         this.varNameToken = varNameToken;
105     }
106
107     public override string ToString()
108     {
109         return string.Format("{0}", varNameToken);
110     }
111 }
112
113 public class BinaryOperationNode : NodeBase
114 {
115     public NodeBase leftNode { get; protected set; }
116     public Token operatorToken { get; protected set; }
117     public NodeBase rightNode { get; protected set; }

```



```

118
119     public BinaryOperationNode(NodeBase leftNode, Token operatorToken,
120         ↪ NodeBase rightNode)
121         : base(leftNode.startPosition, rightNode.endPosition)
122     {
123         this.leftNode = leftNode;
124         this.operatorToken = operatorToken;
125         this.rightNode = rightNode;
126     }
127
128     public override string ToString()
129     {
130         return string.Format("({0}_{1}_{2})", leftNode, operatorToken,
131             ↪ rightNode);
132     }
133
134     public class UnaryOperationNode : NodeBase
135     {
136         public Token operatorToken { get; protected set; }
137         public NodeBase operandNode { get; protected set; }
138
139         public UnaryOperationNode(Token operatorToken, NodeBase operandNode)
140             : base(operatorToken.startPosition, operandNode.endPosition)
141         {
142             this.operatorToken = operatorToken;
143             this.operandNode = operandNode;
144         }
145     }

```

PARSER.CS

Listing 2: Parser.cs

```

1 namespace ShorkSharp
2 {
3     public class Parser
4     {
5         Token[] tokens;
6         int tokenIndex = 0;
7         Token currentToken;
8
9         public Parser(Token[] tokens)
10        {
11            this.tokens = tokens;
12            this.currentToken = this.tokens[0];
13        }
14
15        Token Advance()
16        {
17            tokenIndex++;
18            currentToken = (tokenIndex < tokens.Length) ?
19                ↪ this.tokens[tokenIndex] : null;
20            return currentToken;
21        }
22
23        Token Reverse(int amount = 1)
24        {
25            tokenIndex -= amount;
26            currentToken = (tokenIndex < tokens.Length) ?
27                ↪ this.tokens[tokenIndex] : null;
28            return currentToken;
29        }
30    }

```



```

28
29 public ParseResult Parse()
30 {
31     ParseResult result = ParseStatements();
32
33     if (result.error != null && currentToken.type != TokenType.EOF)
34         return result.Failure(new InvalidSyntaxError("Unexpected_EOF",
35             ↪ currentToken.startPosition));
36
37     return result;
38 }
39
40 //#####
41 protected ParseResult ParseStatements()
42 {
43     ParseResult result = new ParseResult();
44     List<NodeBase> statements = new List<NodeBase>();
45     Position startPosition = currentToken.startPosition.Copy();
46
47     while (currentToken.type != TokenType.NEWLINE)
48     {
49         result.RegisterAdvancement();
50         Advance();
51     }
52
53     NodeBase statement = result.Register(ParseStatement());
54     if (result.error != null)
55         return result;
56     statements.Add(statement);
57
58     bool hasMoreStatements = true;
59     while (true)
60     {
61         int newlineCount = 0;
62         while (currentToken.type == TokenType.NEWLINE)
63         {
64             result.RegisterAdvancement();
65             Advance();
66             newlineCount++;
67         }
68         if (newlineCount == 0)
69             hasMoreStatements = false;
70
71         if (!hasMoreStatements)
72             break;
73
74         statement = result.TryRegister(ParseStatement());
75         if (statement == null)
76         {
77             Reverse(result.toReverseCount);
78             hasMoreStatements = false;
79             continue;
80         }
81         statements.Add(statement);
82     }
83
84     return result.Success(new CodeBlockNode(statements, startPosition,
85         ↪ currentToken.endPosition));
86
87 protected ParseResult ParseStatement()

```



```

88     {
89         throw new NotImplementedException ();
90     }
91
92     protected ParseResult ParseExpression ()
93     {
94         throw new NotImplementedException ();
95     }
96
97     protected ParseResult ParseComparisonExpression ()
98     {
99         throw new NotImplementedException ();
100    }
101
102    protected ParseResult ParseArithmeticExpression ()
103    {
104        throw new NotImplementedException ();
105    }
106
107    protected ParseResult ParseTerm ()
108    {
109        throw new NotImplementedException ();
110    }
111
112    protected ParseResult ParseFactor ()
113    {
114        throw new NotImplementedException ();
115    }
116
117    protected ParseResult ParseExponent ()
118    {
119        throw new NotImplementedException ();
120    }
121
122    protected ParseResult ParseCall ()
123    {
124        throw new NotImplementedException ();
125    }
126
127    protected ParseResult ParseAtom ()
128    {
129        throw new NotImplementedException ();
130    }
131
132    protected ParseResult ParseListExpression ()
133    {
134        throw new NotImplementedException ();
135    }
136
137    // TODO: ParseIfExpression
138
139    protected ParseResult ParseStatement ()
140    {
141        throw new NotImplementedException ();
142    }
143
144    protected ParseResult ParseForExpression ()
145    {
146        throw new NotImplementedException ();
147    }
148
149    protected ParseResult ParseWhileExpression ()

```



```

150     {
151         throw new NotImplementedException();
152     }
153
154     protected ParseResult ParseFunctionDefinition()
155     {
156         throw new NotImplementedException();
157     }
158
159     //#####
160
161     protected delegate ParseResult BinaryOperationDelegate();
162     protected ParseResult ParseBinaryOperation(BinaryOperationDelegate
        ↪ leftFunc, TokenType[] operations)
163     {
164         return ParseBinaryOperation(leftFunc, operations, leftFunc);
165     }
166     protected ParseResult ParseBinaryOperation(BinaryOperationDelegate
        ↪ leftFunc, TokenType[] operations, BinaryOperationDelegate
        ↪ rightFunc)
167     {
168         ParseResult result = new ParseResult();
169
170         NodeBase leftNode = result.Register(leftFunc());
171         if (result.error != null)
172             return result;
173
174         while (operations.Contains(currentToken.type))
175         {
176             Token operatorToken = currentToken;
177             result.RegisterAdvancement();
178             Advance();
179
180             NodeBase rightNode = result.Register(rightFunc());
181             if (result.error != null)
182                 return result;
183
184             leftNode = new BinaryOperationNode(leftNode, operatorToken,
        ↪ rightNode);
185         }
186
187         return result.Success(leftNode);
188     }
189 }
190 }

```

PARSERESULT.CS

Listing 3: ParseResult.cs

```

1 namespace ShorkSharp
2 {
3     public class ParseResult
4     {
5         public ShorkError error { get; protected set; }
6         public NodeBase node { get; protected set; }
7         public int advanceCount { get; protected set; } = 0;
8         public int lastAdvanceCount { get; protected set; } = 0;
9         public int toReverseCount { get; protected set; } = 0;
10
11         public ParseResult() { }
12
13         public void RegisterAdvancement()
14         {

```



```

15         lastAdvanceCount = 1;
16         advanceCount++;
17     }
18
19     public NodeBase Register(ParseResult result)
20     {
21         lastAdvanceCount = result.advanceCount;
22         this.advanceCount += result.advanceCount;
23         if (result.error != null) this.error = result.error;
24         return result.node;
25     }
26
27     public NodeBase TryRegister(ParseResult result)
28     {
29         if (result.error != null)
30         {
31             toReverseCount = result.advanceCount;
32             return null;
33         }
34         return Register(result);
35     }
36
37     public ParseResult Success(NodeBase node)
38     {
39         this.node = node;
40         return this;
41     }
42
43     public ParseResult Failure(ShorkError error)
44     {
45         if (this.error == null || this.lastAdvanceCount == 0)
46             this.error = error;
47         return this;
48     }
49 }
50 }

```

LEXER.CS

Listing 4: Lexer.cs

```

1 namespace ShorkSharp
2 {
3     /// <summary>
4     /// The lexer takes in the input text and converts it into a series of
5     /// tokens.
6     /// </summary>
7     public class Lexer
8     {
9         /// <summary>
10        /// The words recognised as keywords.
11        /// </summary>
12        static readonly string[] KEYWORDS =
13        {
14            "var",
15            "and",
16            "or",
17            "not",
18            "if",
19            "then",
20            "elif",
21            "else",
22            "for",
23            "to",

```



```

23         "step",
24         "func",
25         "while",
26         "do",
27         "end",
28         "return",
29         "continue",
30         "break"
31     };
32     static readonly char[] WHITESPACE = { ' ', '\t', '\r' };
33     static readonly char[] DIGITS = { '0', '1', '2', '3', '4', '5', '6',
    ↪ '7', '8', '9' };
34     static readonly char[] DIGITS_WITH_DOT = DIGITS.Concat(new char[] { '.',
    ↪ }).ToArray();
35     static readonly char[] LETTERS = { 'a', 'b', 'c', 'd', 'e', 'f', 'g',
    ↪ 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't',
    ↪ 'u', 'v', 'w', 'x', 'y', 'z', 'A', 'B', 'C', 'D', 'E', 'F', 'G',
    ↪ 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T',
    ↪ 'U', 'V', 'W', 'X', 'Y', 'Z' };
36     static readonly char[] LETTERS_WITH_UNDERSCORE = LETTERS.Concat(new
    ↪ char[] { '_' }).ToArray();
37
38     public Position position { get; protected set; }
39     public string input { get; protected set; }
40     public char currentChar { get; protected set; } = '\0';
41
42     public Lexer(string input)
43     {
44         this.input = input;
45         this.position = new Position(input);
46     }
47     public Lexer(string input, string filename)
48     {
49         this.input = input;
50         this.position = new Position(filename);
51     }
52
53     void Advance()
54     {
55         position.Advance(currentChar);
56
57         if (position.index < input.Length)
58             currentChar = input[position.index];
59         else
60             currentChar = '\0';
61     }
62
63     /// <summary>
64     /// Runs the lexer and returns the result.
65     /// </summary>
66     /// <returns>If an error occurred, Token[] will be null and ShorkError
    ↪ will contain the error. Otherwise Token[] will contain the tokens
    ↪ and ShorkError will be null.</returns>
67     public (Token[], ShorkError?) Lex()
68     {
69         if (input.Length == 0)
70             return (new Token[] { }, new ShorkError("EmptyInput", "Input
    ↪ text is empty", null));
71         this.currentChar = input[0];
72
73         List<Token> tokens = new List<Token>();
74

```



```

75 while (currentChar != '\0')
76 {
77     if (WHITESPACE.Contains(currentChar))
78     {
79         Advance();
80     }
81
82     // Number Tokens
83     else if (DIGITS.Contains(currentChar))
84     {
85         tokens.Add(MakeNumberToken());
86     }
87
88     // String Tokens
89     else if (currentChar == '"')
90     {
91         (Token token, ShorkError error) = MakeStringToken();
92         if (error != null)
93             return (null, error);
94         tokens.Add(token);
95     }
96
97     // Identifiers and Keywords
98     else if (LETTERS.Contains(currentChar))
99     {
100         tokens.Add(MakeIdentifierToken());
101     }
102
103     // Simple tokens
104     else
105     {
106         switch (currentChar)
107         {
108             default:
109                 return (new Token[] { },
110                        new InvalidCharacterError(string.Format("{0}",
111                                                                currentChar), position));
112
113             case '+':
114                 tokens.Add(new Token(TokenType.PLUS, position));
115                 Advance();
116                 break;
117             case '-':
118                 TokenType ttype = TokenType.MINUS;
119                 Position startPosition = position.Copy();
120                 Advance();
121
122                 if (currentChar == '>')
123                 {
124                     ttype = TokenType.ARROW;
125                     Advance();
126                 }
127
128                 tokens.Add(new Token(ttype, startPosition,
129                                     position));
130                 break;
131             case '*':
132                 tokens.Add(new Token(TokenType.MULTIPLY, position));
133                 Advance();
134                 break;
135             case '/':
136                 tokens.Add(new Token(TokenType.DIVIDE, position));

```



```

134         Advance() ;
135         break;
136     case '^':
137         tokens.Add(new Token(TokenType.EXPONENT, position));
138         Advance() ;
139         break;
140
141     case '!':
142         (Token token, ShorkError error) =
143             ↪ MakeNotEqualsToken();
144         if (error != null) return (null, error);
145         tokens.Add(token);
146         break;
147     case '=':
148         tokens.Add(MakeEqualsToken());
149         break;
150     case '<':
151         tokens.Add(MakeLessThanToken());
152         break;
153     case '>':
154         tokens.Add(MakeGreaterThanToken());
155         break;
156
157     case '.':
158         tokens.Add(new Token(TokenType.DOT, position));
159         Advance() ;
160         break;
161     case ',':
162         tokens.Add(new Token(TokenType.COMMA, position));
163         Advance() ;
164         break;
165
166     case '(':
167         tokens.Add(new Token(TokenType.LPAREN, position));
168         Advance() ;
169         break;
170     case ')':
171         tokens.Add(new Token(TokenType.RPAREN, position));
172         Advance() ;
173         break;
174     case '{':
175         tokens.Add(new Token(TokenType.LBRACE, position));
176         Advance() ;
177         break;
178     case '}':
179         tokens.Add(new Token(TokenType.RBRACE, position));
180         Advance() ;
181         break;
182     case '[':
183         tokens.Add(new Token(TokenType.LBRACKET, position));
184         Advance() ;
185         break;
186     case ']':
187         tokens.Add(new Token(TokenType.RBRACKET, position));
188         Advance() ;
189         break;
190
191     }
192 }
193
194 return (tokens.ToArray(), null);

```



```

195
196 Token MakeNumberToken()
197 {
198     string numstring = string.Empty + currentChar;
199     bool hasDecimalPoint = false;
200     Position startPosition = position.Copy();
201
202     Advance();
203     while (DIGITS_WITH_DOT.Contains(currentChar))
204     {
205         if (currentChar == '.')
206         {
207             if (hasDecimalPoint)
208                 break;
209             else
210                 hasDecimalPoint = true;
211         }
212         numstring += currentChar;
213         Advance();
214     }
215
216     return new Token(TokenType.NUMBER, decimal.Parse(numstring),
217         ↪ startPosition, position);
218 }
219
220 (Token, ShorkError) MakeStringToken()
221 {
222     Position startPosition = position.Copy();
223     string str = string.Empty;
224     Advance();
225
226     bool escaping = false;
227     while (true)
228     {
229         if (escaping)
230         {
231             switch (currentChar)
232             {
233                 default:
234                     return (null, new
235                         ↪ InvalidEscapeSequenceError(string.Format("\\{0}",
236                         ↪ currentChar), position));
237
238                 case '"':
239                     str += '"';
240                     break;
241                 case '\\':
242                     str += '\\';
243                     break;
244                 case 't':
245                     str += '\t';
246                     break;
247             }
248             escaping = false;
249         }
250
251         else if (currentChar == '"')
252         {
253             Advance();
254             break;
255         }
256
257         else if (currentChar == '\\')

```



```

254         escaping = true;
255
256     else
257         str += currentChar;
258
259     Advance();
260 }
261
262     return (new Token(TokenType.STRING, str, startPosition, position),
        ↪ null);
263 }
264
265 Token MakeIdentifierToken()
266 {
267     Position startPosition = position.Copy();
268     string idstr = string.Empty + currentChar;
269     Advance();
270
271     while (LETTERS_WITH_UNDERSCORE.Contains(currentChar))
272     {
273         idstr += currentChar;
274         Advance();
275     }
276
277     if (idstr == "true")
278         return new Token(TokenType.BOOL, true, startPosition, position);
279     else if (idstr == "false")
280         return new Token(TokenType.BOOL, false, startPosition, position);
281     else if (idstr == "null")
282         return new Token(TokenType.NULL, startPosition, position);
283     else
284     {
285         TokenType ttype = KEYWORDS.Contains(idstr.ToLower()) ?
        ↪ TokenType.KEYWORD : TokenType.IDENTIFIER;
286         return new Token(ttype, idstr, startPosition, position);
287     }
288 }
289
290 Token MakeEqualsToken()
291 {
292     Position startPosition = position.Copy();
293     TokenType ttype = TokenType.EQUALS;
294     Advance();
295     if (currentChar == '=')
296     {
297         ttype = TokenType.DOUBLE_EQUALS;
298         Advance();
299     }
300     return new Token(ttype, startPosition, position);
301 }
302
303 (Token, ShorkError) MakeNotEqualsToken()
304 {
305     Position startPosition = position.Copy();
306     Advance();
307     if (currentChar == '=')
308     {
309         Advance();
310         return (new Token(TokenType.NOT_EQUALS, startPosition,
        ↪ position), null);
311     }
312     return (null, new InvalidCharacterError("", position));

```



```

313     }
314
315     Token MakeLessThanToken()
316     {
317         Position startPosition = position.Copy();
318         TokenType ttype = TokenType.LESS_THAN;
319         Advance();
320         if (currentChar == '=')
321         {
322             ttype = TokenType.LESS_THAN_OR_EQUAL;
323             Advance();
324         }
325         return new Token(ttype, startPosition, position);
326     }
327
328     Token MakeGreaterThanToken()
329     {
330         Position startPosition = position.Copy();
331         TokenType ttype = TokenType.GREATER_THAN;
332         Advance();
333         if (currentChar == '=')
334         {
335             ttype = TokenType.GREATER_THAN_OR_EQUAL;
336             Advance();
337         }
338         return new Token(ttype, startPosition, position);
339     }
340 }
341 }

```

SHORKERROR.CS

Listing 5: ShorkError.cs

```

1 namespace ShorkSharp
2 {
3     public class ShorkError
4     {
5         public string errorName { get; protected set; }
6         public string details { get; protected set; }
7
8         public Position startPosition { get; protected set; }
9
10        public ShorkError(string errorName, string details, Position
11            ↪ startPosition)
12        {
13            this.errorName = errorName;
14            this.details = details;
15            this.startPosition = startPosition;
16        }
17
18        public override string ToString()
19        {
20            string output = string.Format("{0}:_{1}", errorName, details);
21
22            if (startPosition != null)
23                output += string.Format("\nFile:_{0}',_{1}line_{1}",
24                    ↪ startPosition.filename, startPosition.line+1);
25
26            return output;
27        }
28    }
29
30    public class InvalidCharacterError : ShorkError

```



```

29     {
30         public InvalidCharacterError(string details, Position startPosition)
31             : base("Invalid Character", details, startPosition) { }
32     }
33
34     public class InvalidSyntaxError : ShorkError
35     {
36         public InvalidSyntaxError(string details, Position startPosition)
37             : base("Invalid Syntax", details, startPosition) { }
38     }
39
40     public class InvalidEscapeSequenceError : ShorkError
41     {
42         public InvalidEscapeSequenceError(string details, Position startPosition)
43             : base("Invalid Escape Sequence", details, startPosition) { }
44     }
45 }

```

TOKEN.CS

Listing 6: Token.cs

```

1 namespace ShorkSharp
2 {
3     public class Token
4     {
5         public TokenType type { get; protected set; }
6         public dynamic value { get; protected set; }
7
8         public Position startPosition { get; protected set; }
9         public Position endPosition { get; protected set; }
10
11         public Token(TokenType type, Position startPosition)
12         {
13             this.type = type;
14             this.value = null;
15             this.startPosition = startPosition.Copy();
16             this.endPosition = startPosition.Copy();
17         }
18         public Token(TokenType type, Position startPosition, Position
19             ↪ endPosition)
20         {
21             this.type = type;
22             this.value = null;
23             this.startPosition = startPosition.Copy();
24             this.endPosition = endPosition.Copy();
25         }
26         public Token(TokenType type, dynamic value, Position startPosition)
27         {
28             this.type = type;
29             this.value = value;
30             this.startPosition = startPosition.Copy();
31             this.endPosition = startPosition.Copy();
32         }
33         public Token(TokenType type, dynamic value, Position startPosition,
34             ↪ Position endPosition)
35         {
36             this.type = type;
37             this.value = value;
38             this.startPosition = startPosition.Copy();
39             this.endPosition = endPosition.Copy();
40         }
41
42         public override string ToString()

```



```

41     {
42         if (value == null)
43             return string.Format("[{0}]", type);
44         else
45             return string.Format("[{0}]:{1}", type, value);
46     }
47 }
48 }

```

TOKENTYPE.CS

Listing 7: TokenType.cs

```

1 namespace ShorkSharp
2 {
3     public enum TokenType
4     {
5         NUMBER,
6         STRING,
7         BOOL,
8         NULL,
9
10        KEYWORD,
11        IDENTIFIER,
12
13        PLUS,
14        MINUS,
15        MULTIPLY,
16        DIVIDE,
17        EXPONENT,
18
19        EQUALS,
20        DOUBLE_EQUALS,
21        NOT_EQUALS,
22        LESS_THAN,
23        GREATER_THAN,
24        LESS_THAN_OR_EQUAL,
25        GREATER_THAN_OR_EQUAL,
26
27        DOT,
28        COMMA,
29        ARROW,
30
31        LPAREN,
32        RPAREN,
33        LBRACE,
34        RBRACE,
35        LBRACKET,
36        RBRACKET,
37
38        NEWLINE,
39        EOF
40    }
41 }

```