

Shork#

Miss Ylva Llywelyn

2023/10/14

Based on the tutorial series by David Callanan.

CONTENTS

CH. 1: GRAMMAR.....	1	ParseResult.cs	12
CH. 2: CODE LISTING	2	Lexer.cs	13
NodeBase.cs	2	ShorkError.cs	18
Parser.cs.....	6	Token.cs	19
		TokenType.cs.....	20

CHAPTER 1: 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>	NEWLINE* <i>statement</i> (NEWLINE+ <i>statement</i>)* NEWLINE*
<i>statement</i>	KEYWORD:RETURN <i>expression</i> ? KEYWORD:CONTINUE KEYWORD:BREAK <i>expression</i>
<i>expression</i>	KEYWORD:VAR IDENTIFIER = <i>expression</i> <i>comparision_expression</i> ((KEYWORD:AND KEYWORD:OR) <i>comparision_expression</i>)*
<i>comparision_expression</i>	KEYWORD:NOT <i>comparision_expression</i> <i>arithmetic_expression</i> ((== != < <= > >=) <i>arithmetic_expression</i>)*
<i>arithmetic_expression</i>	<i>term</i> ((+ -) <i>term</i>)*
<i>term</i>	<i>factor</i> ((* /) <i>factor</i>)*
<i>factor</i>	(+ -)? <i>factor</i> <i>exponent</i>
<i>exponent</i>	<i>call</i> (^ <i>factor</i>)*

CHAPTER 2: CODE LISTING

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
32     public class NumberNode : NodeBase
33     {
34         public Token numToken { get; protected set; }
35
36         public NumberNode(Token numToken)
37             : base(numToken.startPosition, numToken.endPosition)
38         {
39             this.numToken = numToken;
40         }
41
42         public override string ToString()
43         {
44             return string.Format("({0})", numToken);
45         }
46     }
47
48     public class StringNode : NodeBase
49     {
50         public Token strToken { get; protected set; }
51
52         public StringNode(Token strToken)
53             : base(strToken.startPosition, strToken.endPosition)
54         {
55             this.strToken = strToken;
```



```

55     }
56
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 public class VarAssignNode : NodeBase
79 {
80     public Token varNameToken { get; protected set; }
81     public NodeBase valueNode { get; protected set; }
82
83     public VarAssignNode(Token varNameToken, NodeBase valueNode)
84         : base(varNameToken.startPosition, valueNode.endPosition)
85     {
86         this.varNameToken = varNameToken;
87         this.valueNode = valueNode;
88     }
89
90     public override string ToString()
91     {
92         return string.Format("{0}={1}", varNameToken, valueNode);
93     }
94 }
95
96 public class VarAccessNode : NodeBase
97 {
98     public Token varNameToken { get; protected set; }
99
100     public VarAccessNode(Token varNameToken)
101         : base(varNameToken.startPosition, varNameToken.endPosition)
102     {
103         this.varNameToken = varNameToken;
104     }
105
106     public override string ToString()
107     {
108         return string.Format("{0}", varNameToken);
109     }
110 }
111
112 public class BinaryOperationNode : NodeBase
113 {
114     public NodeBase leftNode { get; protected set; }
115

```



```

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     }
146
147     public class IfNode : NodeBase
148     {
149         public (NodeBase, NodeBase)[] caseNodes { get; protected set; }
150         public NodeBase elseNode { get; protected set; }
151
152         public IfNode((NodeBase, NodeBase)[] caseNodes)
153             : base(caseNodes[0].Item1.startPosition,
154                 ↪ caseNodes[^1].Item2.endPosition)
155         {
156             this.caseNodes = caseNodes;
157         }
158         public IfNode((NodeBase, NodeBase)[] caseNodes, NodeBase elseNode)
159             : base(caseNodes[0].Item1.startPosition, elseNode.endPosition)
160         {
161             this.caseNodes = caseNodes;
162             this.elseNode = elseNode;
163         }
164     }
165
166     public class ForNode : NodeBase
167     {
168         public Token varNameToken { get; protected set; }
169         public NodeBase startValueNode { get; protected set; }
170         public NodeBase endValueNode { get; protected set; }
171         public NodeBase stepValueNode { get; protected set; }
172         public NodeBase bodyNode { get; protected set; }
173         public bool shouldReturnNull { get; protected set; }
174
175         public ForNode(Token varNameToken,
176             NodeBase startValueNode,

```



```

175         NodeBase endValueNode,
176         NodeBase stepValueNode,
177         NodeBase bodyNode,
178         bool shouldReturnNull)
179     : base(varNameToken.startPosition, bodyNode.endPosition)
180     {
181         this.varNameToken = varNameToken;
182         this.startValueNode = startValueNode;
183         this.endValueNode = endValueNode;
184         this.stepValueNode = stepValueNode;
185         this.bodyNode = bodyNode;
186         this.shouldReturnNull = shouldReturnNull;
187     }
188 }
189
190 public class WhileNode : NodeBase
191 {
192     public NodeBase conditionNode { get; protected set; }
193     public NodeBase bodyNode { get; protected set; }
194     public bool shouldReturnNull { get; protected set; }
195
196     public WhileNode(NodeBase conditionNode, NodeBase bodyNode, bool
197         ↪ shouldReturnNull)
198         : base(conditionNode.startPosition, bodyNode.endPosition)
199     {
200         this.conditionNode = conditionNode;
201         this.bodyNode = bodyNode;
202         this.shouldReturnNull = shouldReturnNull;
203     }
204 }
205
206 public class FunctionDefinitionNode : NodeBase
207 {
208     public Token varNameToken { get; protected set; }
209     public Token[] argNameTokens { get; protected set; }
210     public NodeBase bodyNode { get; protected set; }
211     public bool shouldAutoReturn { get; protected set; }
212
213     public FunctionDefinitionNode(Token varNameToken,
214         Token[] argNameTokens,
215         NodeBase bodyNode,
216         bool shouldAutoReturn)
217         : base(varNameToken.startPosition, bodyNode.endPosition)
218     {
219         this.varNameToken = varNameToken;
220         this.argNameTokens = argNameTokens;
221         this.bodyNode = bodyNode;
222         this.shouldAutoReturn = shouldAutoReturn;
223     }
224 }
225
226 public class CallNode : NodeBase
227 {
228     public NodeBase nodeToCall { get; protected set; }
229     public NodeBase[] argumentNodes { get; protected set; }
230
231     public CallNode(NodeBase nodeToCall, NodeBase[] argumentNodes)
232         : base(nodeToCall.startPosition, (argumentNodes.Length > 0) ?
233             ↪ argumentNodes[1].endPosition : nodeToCall.endPosition)
234     {
235         this.nodeToCall = nodeToCall;
236         this.argumentNodes = argumentNodes;

```



```

235     }
236 }
237
238 public class ReturnNode : NodeBase
239 {
240     public NodeBase nodeToReturn { get; protected set; }
241
242     public ReturnNode(Position startPosition, Position endPosition)
243         : base(startPosition, endPosition) { }
244     public ReturnNode(NodeBase nodeToReturn)
245         : base(nodeToReturn.startPosition, nodeToReturn.endPosition)
246     {
247         this.nodeToReturn = nodeToReturn;
248     }
249 }
250
251 public class ContinueNode : NodeBase
252 {
253     public ContinueNode(Position startPosition, Position endPosition)
254         : base(startPosition, endPosition) { }
255 }
256
257 public class BreakNode : NodeBase
258 {
259     public BreakNode(Position startPosition, Position endPosition)
260         : base(startPosition, endPosition) { }
261 }
262 }

```

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
31        public ParseResult Parse()
32        {

```



```

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
42 protected ParseResult ParseStatements()
43 {
44     ParseResult result = new ParseResult();
45     List<NodeBase> statements = new List<NodeBase>();
46     Position startPosition = currentToken.startPosition.Copy();
47
48     while (currentToken.type != TokenType.NEWLINE)
49     {
50         result.RegisterAdvancement();
51         Advance();
52     }
53
54     NodeBase statement = result.Register(ParseStatement());
55     if (result.error != null)
56         return result;
57     statements.Add(statement);
58
59     bool hasMoreStatements = true;
60     while (true)
61     {
62         int newlineCount = 0;
63         while (currentToken.type == TokenType.NEWLINE)
64         {
65             result.RegisterAdvancement();
66             Advance();
67             newlineCount++;
68         }
69         if (newlineCount == 0)
70             hasMoreStatements = false;
71
72         if (!hasMoreStatements)
73             break;
74
75         statement = result.TryRegister(ParseStatement());
76         if (statement == null)
77         {
78             Reverse(result.toReverseCount);
79             hasMoreStatements = false;
80             continue;
81         }
82         statements.Add(statement);
83     }
84
85     return result.Success(new CodeBlockNode(statements, startPosition,
86         ↪ currentToken.endPosition));
87 }
88
89 protected ParseResult ParseStatement()
90 {
91     ParseResult result = new ParseResult();
92     Position startPosition = currentToken.startPosition.Copy();

```



```

91
92     if (currentToken.Matches(TokenType.KEYWORD, "return"))
93     {
94         result.RegisterAdvancement();
95         Advance();
96
97         NodeBase expression = result.TryRegister(ParseExpression());
98         if (expression == null)
99         {
100             Reverse(result.toReverseCount);
101             return result.Success(new ReturnNode(startPosition,
102                 ↪ currentToken.endPosition));
103         }
104         else
105             return result.Success(new ReturnNode(expression));
106     }
107
108     else if (currentToken.Matches(TokenType.KEYWORD, "continue"))
109     {
110         result.RegisterAdvancement();
111         Advance();
112         return result.Success(new ContinueNode(startPosition,
113             ↪ currentToken.endPosition));
114     }
115
116     else if (currentToken.Matches(TokenType.KEYWORD, "break"))
117     {
118         result.RegisterAdvancement();
119         Advance();
120         return result.Success(new BreakNode(startPosition,
121             ↪ currentToken.endPosition));
122     }
123
124     else
125     {
126         NodeBase expression = result.Register(ParseExpression());
127         if (result.error != null)
128             return result.Failure(new InvalidSyntaxError("Expected_
129                 ↪ 'RETURN',_ 'CONTINUE',_ 'BREAK',_ 'VAR',_ 'IF',_ 'FOR',_
130                 ↪ 'WHILE',_ 'FUN',_ int,_ float,_ identifier,_ '+',_ '-',_ '(',_
131                 ↪ '['_ or_ 'NOT'", currentToken.startPosition));
132
133         return result.Success(expression);
134     }
135 }
136
137 protected ParseResult ParseExpression()
138 {
139     ParseResult result = new ParseResult();
140
141     if (currentToken.Matches(TokenType.KEYWORD, "var"))
142     {
143         result.RegisterAdvancement();
144         Advance();
145
146         if (currentToken.type != TokenType.IDENTIFIER)
147             return result.Failure(new InvalidSyntaxError("Expected_
148                 ↪ identifier", currentToken.startPosition));
149
150         Token varNameToken = currentToken;
151         result.RegisterAdvancement();
152         Advance();

```



```

146
147         if (currentToken.type != TokenType.EQUALS)
148             return result.Failure(new InvalidSyntaxError("Expected '='",
149                 ↪ currentToken.startPosition));
150
151     result.RegisterAdvancement();
152     Advance();
153
154     NodeBase expression = result.Register(ParseExpression());
155     if (result.error != null) return result;
156     return result.Success(new VarAssignNode(varNameToken,
157         ↪ expression));
158
159 else
160 {
161     NodeBase node =
162         ↪ result.Register(ParseBinaryOperation(ParseComparisonExpression,
163             new
164                 ↪ (TokenType,
165                 ↪ string) []
166                 ↪ {
167                     (TokenType.KEYWORD, "and"),
168                     (TokenType.KEYWORD, "or")
169                 }));
170
171     if (result.error != null)
172         return result.Failure(new InvalidSyntaxError("Expected
173             ↪ 'VAR', 'IF', 'FOR', 'WHILE', 'FUNC', number,
174             ↪ identifier, '+', '-', '(', '[', or 'NOT'",
175             ↪ currentToken.startPosition));
176     return result.Success(node);
177 }
178
179 protected ParseResult ParseComparisonExpression()
180 {
181     ParseResult result = new ParseResult();
182     NodeBase node;
183
184     if (currentToken.Matches(TokenType.KEYWORD, "not"))
185     {
186         Token operatorToken = currentToken;
187         result.RegisterAdvancement();
188         Advance();
189
190         node = result.Register(ParseComparisonExpression());
191         if (result.error != null) return result;
192         return result.Success(node);
193     }
194
195     node =
196         ↪ result.Register(ParseBinaryOperation(ParseArithmeticExpression,
197         ↪ new TokenType[]

```



```

195         if (result.error != null)
196             return result.Failure(new InvalidSyntaxError("Expected_, number, _
197                 ↳ identifier, _, '+', _, '-', _, '(', _, '[', _, 'IF', _, 'FOR', _, 'WHILE', _,
                 ↳ 'FUNC', _or_ 'NOT'", currentToken.startPosition));
198         return result.Success(node);
199     }
200
201     protected ParseResult ParseArithmeticExpression()
202     {
203         throw new NotImplementedException();
204     }
205
206     protected ParseResult ParseTerm()
207     {
208         throw new NotImplementedException();
209     }
210
211     protected ParseResult ParseFactor()
212     {
213         throw new NotImplementedException();
214     }
215
216     protected ParseResult ParseExponent()
217     {
218         throw new NotImplementedException();
219     }
220
221     protected ParseResult ParseCall()
222     {
223         throw new NotImplementedException();
224     }
225
226     protected ParseResult ParseAtom()
227     {
228         throw new NotImplementedException();
229     }
230
231     protected ParseResult ParseListExpression()
232     {
233         throw new NotImplementedException();
234     }
235
236     /* TODO: ParseIfExpression
237
238     protected ParseResult ParseStatement()
239     {
240         throw new NotImplementedException();
241     } */
242
243     protected ParseResult ParseForExpression()
244     {
245         throw new NotImplementedException();
246     }
247
248     protected ParseResult ParseWhileExpression()
249     {
250         throw new NotImplementedException();
251     }
252
253     protected ParseResult ParseFunctionDefinition()
254     {

```



```

255         throw new NotImplementedException();
256     }
257
258     //#####
259
260     protected delegate ParseResult BinaryOperationDelegate();
261     protected ParseResult ParseBinaryOperation(BinaryOperationDelegate
        ↪ leftFunc, TokenType[] operations)
262     {
263         return ParseBinaryOperation(leftFunc, operations, leftFunc);
264     }
265     protected ParseResult ParseBinaryOperation(BinaryOperationDelegate
        ↪ leftFunc, TokenType[] operations, BinaryOperationDelegate
        ↪ rightFunc)
266     {
267         ParseResult result = new ParseResult();
268
269         NodeBase leftNode = result.Register(leftFunc());
270         if (result.error != null)
271             return result;
272
273         while (operations.Contains(currentToken.type))
274         {
275             Token operatorToken = currentToken;
276             result.RegisterAdvancement();
277             Advance();
278
279             NodeBase rightNode = result.Register(rightFunc());
280             if (result.error != null)
281                 return result;
282
283             leftNode = new BinaryOperationNode(leftNode, operatorToken,
                ↪ rightNode);
284         }
285
286         return result.Success(leftNode);
287     }
288     protected ParseResult ParseBinaryOperation(BinaryOperationDelegate
        ↪ leftFunc, (TokenType, string)[] operations)
289     {
290         return ParseBinaryOperation(leftFunc, operations, leftFunc);
291     }
292     protected ParseResult ParseBinaryOperation(BinaryOperationDelegate
        ↪ leftFunc, (TokenType, string)[] operations,
        ↪ BinaryOperationDelegate rightFunc)
293     {
294         ParseResult result = new ParseResult();
295
296         NodeBase leftNode = result.Register(leftFunc());
297         if (result.error != null)
298             return result;
299
300         while (operations.Contains((currentToken.type,
            ↪ (string)currentToken.value)))
301         {
302             Token operatorToken = currentToken;
303             result.RegisterAdvancement();
304             Advance();
305
306             NodeBase rightNode = result.Register(rightFunc());
307             if (result.error != null)
308                 return result;

```



```

309         leftNode = new BinaryOperationNode(leftNode, operatorToken,
310             ↪ rightNode);
311     }
312
313     return result.Success(leftNode);
314 }
315 }
316 }

```

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

```
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",
24            "step",
25            "func",
26            "while",
27            "do",
28            "end",
29            "return",
30            "continue",
31            "break"
32        };
33        static readonly char[] WHITESPACE = { ' ', '\t', '\r' };
34        static readonly char[] DIGITS = { '0', '1', '2', '3', '4', '5', '6',
35            ↪ '7', '8', '9' };
36        static readonly char[] DIGITS_WITH_DOT = DIGITS.Concat(new char[] { '.',
37            ↪ }) .ToArray();
38        static readonly char[] LETTERS = { 'a', 'b', 'c', 'd', 'e', 'f', 'g',
39            ↪ 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't',
40            ↪ 'u', 'v', 'w', 'x', 'y', 'z', 'A', 'B', 'C', 'D', 'E', 'F', 'G',
41            ↪ 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T',
42            ↪ 'U', 'V', 'W', 'X', 'Y', 'Z' };
43        static readonly char[] LETTERS_WITH_UNDERSCORE = LETTERS.Concat(new
44            ↪ char[] { '_' }) .ToArray();
45
46        public Position position { get; protected set; }
47        public string input { get; protected set; }
48        public char currentChar { get; protected set; } = '\0';
49
50        public Lexer(string input)
51        {
52            this.input = input;
53            this.position = new Position(input);
54        }
55        public Lexer(string input, string filename)
56        {
57            this.input = input;
58            this.position = new Position(filename);
59        }
60    }
61 }
```



```

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
67     ↳ will contain the error. Otherwise Token[] will contain the tokens
68     ↳ and ShorkError will be null.</returns>
69 public (Token[], ShorkError?) Lex()
70 {
71     if (input.Length == 0)
72         return (new Token[] { }, new ShorkError("EmptyInput", "Input_
73             ↳ text_is_empty", null));
74     this.currentChar = input[0];
75
76     List<Token> tokens = new List<Token>();
77
78     while (currentChar != '\0')
79     {
80         if (WHITESPACE.Contains(currentChar))
81         {
82             Advance();
83         }
84
85         // Number Tokens
86         else if (DIGITS.Contains(currentChar))
87         {
88             tokens.Add(MakeNumberToken());
89         }
90
91         // String Tokens
92         else if (currentChar == '"')
93         {
94             (Token token, ShorkError error) = MakeStringToken();
95             if (error != null)
96                 return (null, error);
97             tokens.Add(token);
98         }
99
100         // Identifiers and Keywords
101         else if (LETTERS.Contains(currentChar))
102         {
103             tokens.Add(MakeIdentifierToken());
104         }
105
106         // Simple tokens
107         else
108         {
109             switch (currentChar)
110             {
111                 default:
112                     return (new Token[] { },
113                             new
114                                 ↳ InvalidCharacterError(string.Format("'{0}'",

```



```

111         ↪ currentChar), position));
112     case '+':
113         tokens.Add(new Token(TokenType.PLUS, position));
114         Advance();
115         break;
116     case '-':
117         TokenType ttype = TokenType.MINUS;
118         Position startPosition = position.Copy();
119         Advance();
120
121         if (currentChar == '>')
122         {
123             ttype = TokenType.ARROW;
124             Advance();
125         }
126
127         tokens.Add(new Token(ttype, startPosition,
128             ↪ position));
129         break;
130     case '*':
131         tokens.Add(new Token(TokenType.MULTIPLY, position));
132         Advance();
133         break;
134     case '/':
135         tokens.Add(new Token(TokenType.DIVIDE, position));
136         Advance();
137         break;
138     case '^':
139         tokens.Add(new Token(TokenType.EXPONENT, position));
140         Advance();
141         break;
142
143     case '!':
144         (Token token, ShorkError error) =
145             ↪ MakeNotEqualsToken();
146         if (error != null) return (null, error);
147         tokens.Add(token);
148         break;
149     case '=':
150         tokens.Add(MakeEqualsToken());
151         break;
152     case '<':
153         tokens.Add(MakeLessThanToken());
154         break;
155     case '>':
156         tokens.Add(MakeGreaterThanToken());
157         break;
158
159     case '.':
160         tokens.Add(new Token(TokenType.DOT, position));
161         Advance();
162         break;
163     case ',':
164         tokens.Add(new Token(TokenType.COMMA, position));
165         Advance();
166         break;
167
168     case '(':
169         tokens.Add(new Token(TokenType.LPAREN, position));
170         Advance();
171         break;
172     case ')':

```



```

170         tokens.Add(new Token(TokenType.RPAREN, position));
171         Advance();
172         break;
173     case '{':
174         tokens.Add(new Token(TokenType.LBRACE, position));
175         Advance();
176         break;
177     case '}':
178         tokens.Add(new Token(TokenType.RBRACE, position));
179         Advance();
180         break;
181     case '[':
182         tokens.Add(new Token(TokenType.LBRACKET, position));
183         Advance();
184         break;
185     case ']':
186         tokens.Add(new Token(TokenType.RBRACKET, position));
187         Advance();
188         break;
189     }
190 }
191 }
192
193 return (tokens.ToArray(), null);
194 }
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)

```



```

231         {
232             default:
233                 return (null, new
                    ↳ InvalidEscapeSequenceError(string.Format("\\{0}",
                    ↳ currentChar), position));
234             case "'":
235                 str += "'";
236                 break;
237             case "\\":
238                 str += "\\ ";
239                 break;
240             case "t":
241                 str += "\t ";
242                 break;
243         }
244         escaping = false;
245     }
246
247     else if (currentChar == "'")
248     {
249         Advance();
250         break;
251     }
252
253     else if (currentChar == "\\ ")
254         escaping = true;
255
256     else
257         str += currentChar;
258
259     Advance();
260 }
261
262 return (new Token(Token.Type.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(Token.Type.BOOL, true, startPosition, position);
279     else if (idstr == "false")
280         return new Token(Token.Type.BOOL, false, startPosition, position);
281     else if (idstr == "null")
282         return new Token(Token.Type.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,
311             ↪ position), null);
312     }
313     return (null, new InvalidCharacterError("", position));
314 }
315
316 Token MakeLessThanToken()
317 {
318     Position startPosition = position.Copy();
319     TokenType ttype = TokenType.LESS_THAN;
320     Advance();
321     if (currentChar == '<')
322     {
323         ttype = TokenType.LESS_THAN_OR_EQUAL;
324         Advance();
325     }
326     return new Token(ttype, startPosition, position);
327 }
328
329 Token MakeGreaterThanToken()
330 {
331     Position startPosition = position.Copy();
332     TokenType ttype = TokenType.GREATER_THAN;
333     Advance();
334     if (currentChar == '>')
335     {
336         ttype = TokenType.GREATER_THAN_OR_EQUAL;
337         Advance();
338     }
339     return new Token(ttype, startPosition, position);
340 }
341 }

```

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
        ↳ startPosition)
11    {
12        this.errorName = errorName;
13        this.details = details;
14        this.startPosition = startPosition;
15    }
16
17    public override string ToString()
18    {
19        string output = string.Format("{0}:_{1}", errorName, details);
20
21        if (startPosition != null)
22            output += string.Format("\nFile:_{0}',_{1}line_{1}",
                ↳ startPosition.filename, startPosition.line+1);
23
24        return output;
25    }
26 }
27
28 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

```

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
        ↳ endPosition)

```



```

19     {
20         this.type = type;
21         this.value = null;
22         this.startPosition = startPosition.Copy();
23         this.endPosition = endPosition.Copy();
24     }
25     public Token(TokenType type, dynamic value, Position startPosition)
26     {
27         this.type = type;
28         this.value = value;
29         this.startPosition = startPosition.Copy();
30         this.endPosition = startPosition.Copy();
31     }
32     public Token(TokenType type, dynamic value, Position startPosition,
33         ↪ Position endPosition)
34     {
35         this.type = type;
36         this.value = value;
37         this.startPosition = startPosition.Copy();
38         this.endPosition = endPosition.Copy();
39     }
40     public bool Matches(TokenType type)
41     {
42         return this.type == type;
43     }
44     public bool Matches(TokenType type, dynamic value)
45     {
46         if (type == TokenType.KEYWORD)
47             return this.type == type && ((string)this.value).ToLower() ==
48             ↪ ((string)value).ToLower();
49         return this.type == type && this.value == value;
50     }
51     public override string ToString()
52     {
53         if (value == null)
54             return string.Format("[{0}]", type);
55         else
56             return string.Format("[{0}_:{1}]", type, value);
57     }
58 }
59 }

```

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 }
```