

Shork#

Miss Ylva Llywelyn

2023/10/14

Based on the tutorial series by David Callanan.

CONTENTS

CH. 1: GRAMMAR.....	1	ParseResult.cs	18
CH. 2: CODE LISTING	2	Lexer.cs	19
NodeBase.cs	2	ShorkError.cs	25
Parser.cs.....	6	Token.cs	25
		TokenType.cs.....	27

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;
56         }
57     }
58 }
```



```

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         if (currentToken.type != TokenType.EQUALS)
147             return result.Failure(new InvalidSyntaxError("Expected '=',",
148                 ↪ currentToken.startPosition));
149
150         result.RegisterAdvancement();
151         Advance();
152
153         NodeBase expression = result.Register(ParseExpression());
154         if (result.error != null) return result;
155         return result.Success(new VarAssignNode(varNameToken,
156             ↪ expression));
157     }
158     else
159     {
160         NodeBase node =
161             ↪ result.Register(ParseBinaryOperation(ParseComparisonExpression,
162                 ↪ new TokenType[] { TokenType.KEYWORD, "and",
163                 ↪ (TokenType.KEYWORD, "or") }));
164         if (result.error != null)
165             return result.Failure(new InvalidSyntaxError("Expected
166                 ↪ 'VAR', 'IF', 'FOR', 'WHILE', 'FUNC', number,
167                 ↪ identifier, '+', '-', '(', '[', 'or 'NOT'",
168                 ↪ currentToken.startPosition));
169         return result.Success(node);
170     }
171 }
172
173 protected ParseResult ParseComparisonExpression()
174 {
175     ParseResult result = new ParseResult();
176     NodeBase node;
177
178     if (currentToken.Matches(TokenType.KEYWORD, "not"))
179     {
180         Token operatorToken = currentToken;
181         result.RegisterAdvancement();
182         Advance();
183
184         node = result.Register(ParseComparisonExpression());
185         if (result.error != null) return result;
186         return result.Success(node);
187     }
188
189     node =
190         ↪ result.Register(ParseBinaryOperation(ParseArithmeticExpression,
191         ↪ new TokenType[] { TokenType.DOUBLE_EQUALS,
192         ↪ TokenType.NOT_EQUALS, TokenType.LESS_THAN,
193         ↪ TokenType.GREATER_THAN, TokenType.LESS_THAN_OR_EQUAL,
194         ↪ TokenType.GREATER_THAN_OR_EQUAL }));
195     if (result.error != null)
196         return result.Failure(new InvalidSyntaxError("Expected number,
197             ↪ identifier, '+', '-', '(', '[', 'IF', 'FOR', 'WHILE',
198             ↪ 'FUNC', 'or 'NOT'", currentToken.startPosition));
199     return result.Success(node);
200 }
201
202 protected ParseResult ParseArithmeticExpression()
203 {
204     return ParseBinaryOperation(ParseTerm, new TokenType[] {
205         ↪ TokenType.PLUS, TokenType.MINUS });
206 }

```



```

192     }
193
194     protected ParseResult ParseTerm()
195     {
196         return ParseBinaryOperation(ParseFactor, new TokenType[] {
197             ↳ TokenType.MULTIPLY, TokenType.DIVIDE });
198     }
199
200     protected ParseResult ParseFactor()
201     {
202         ParseResult result = new ParseResult();
203
204         if (currentToken.Matches(TokenType.PLUS, TokenType.MINUS))
205         {
206             Token operandToken = currentToken;
207             result.RegisterAdvancement();
208             Advance();
209             NodeBase factor = result.Register(ParseFactor());
210             if (result.error != null) return result;
211             return result.Success(new UnaryOperationNode(operandToken,
212                 ↳ factor));
213         }
214
215         return ParseExponent();
216     }
217
218     protected ParseResult ParseExponent()
219     {
220         return ParseBinaryOperation(ParseCall, new TokenType[] {
221             ↳ TokenType.EXPONENT }, ParseFactor);
222     }
223
224     protected ParseResult ParseCall()
225     {
226         ParseResult result = new ParseResult();
227
228         NodeBase atom = result.Register(ParseAtom());
229         if (result.error != null) return result;
230
231         if (currentToken.type == TokenType.LPAREN)
232         {
233             result.RegisterAdvancement();
234             Advance();
235
236             List<NodeBase> args = new List<NodeBase>();
237
238             if (currentToken.type == TokenType.RPAREN)
239             {
240                 result.RegisterAdvancement();
241                 Advance();
242             }
243             else
244             {
245                 args.Add(result.Register(ParseExpression()));
246                 if (result.error != null)
247                     return result.Failure(new InvalidSyntaxError("Expected
248                         ↳ ')', 'VAR', 'IF', 'FOR', 'WHILE', 'FUNC', 'number',
249                         ↳ identifier, '+', '-', '(', '[', 'or 'NOT'",
250                         ↳ currentToken.startPosition));
251             }
252
253             while (currentToken.type == TokenType.COMMA)
254             {

```



```

248         result.RegisterAdvancement();
249         Advance();
250
251         args.Add(result.Register(ParseExpression()));
252         if (result.error != null) return result;
253     }
254
255     if (currentToken.type != TokenType.RPAREN)
256         return result.Failure(new InvalidSyntaxError("Expected␣
           ↳ ', ' or ')' ", currentToken.startPosition));
257
258     result.RegisterAdvancement();
259     Advance();
260 }
261
262     return result.Success(new CallNode(atom, args.ToArray()));
263 }
264 return result.Success(atom);
265 }
266
267 protected ParseResult ParseAtom()
268 {
269     ParseResult result = new ParseResult();
270
271     if (currentToken.type == TokenType.NUMBER)
272     {
273         result.RegisterAdvancement();
274         Advance();
275         return result.Success(new NumberNode(currentToken));
276     }
277
278     else if (currentToken.type == TokenType.STRING)
279     {
280         result.RegisterAdvancement();
281         Advance();
282         return result.Success(new StringNode(currentToken));
283     }
284
285     else if (currentToken.type == TokenType.IDENTIFIER)
286     {
287         result.RegisterAdvancement();
288         Advance();
289         return result.Success(new VarAccessNode(currentToken));
290     }
291
292     else if (currentToken.type == TokenType.LPAREN)
293     {
294         result.RegisterAdvancement();
295         Advance();
296
297         NodeBase expression = result.Register(ParseExpression());
298         if (result.error != null) return result;
299
300         if (currentToken.type == TokenType.RPAREN)
301         {
302             result.RegisterAdvancement();
303             Advance();
304             return result.Success(expression);
305         }
306         else return result.Failure(new InvalidSyntaxError("Expected␣
           ↳ ')' ", currentToken.startPosition));
307     }

```



```

308
309     else if (currentToken.type == TokenType.LBRACKET)
310     {
311         NodeBase list = result.Register(ParseListExpression());
312         if (result.error != null) return result;
313         return result.Success(list);
314     }
315
316     else if (currentToken.Matches(TokenType.KEYWORD, "if"))
317     {
318         NodeBase ifNode = result.Register(ParseIfExpression());
319         if (result.error != null) return result;
320         return result.Success(ifNode);
321     }
322
323     else if (currentToken.Matches(TokenType.KEYWORD, "for"))
324     {
325         NodeBase forNode = result.Register(ParseForExpression());
326         if (result.error != null) return result;
327         return result.Success(forNode);
328     }
329
330     else if (currentToken.Matches(TokenType.KEYWORD, "while"))
331     {
332         NodeBase whileNode = result.Register(ParseWhileExpression());
333         if (result.error != null) return result;
334         return result.Success(whileNode);
335     }
336
337     else if (currentToken.Matches(TokenType.KEYWORD, "func"))
338     {
339         NodeBase functionDefinition =
340             ↪ result.Register(ParseFunctionDefinition());
341         if (result.error != null) return result;
342         return result.Success(functionDefinition);
343     }
344
345     else return result.Failure(new InvalidSyntaxError("Expected number, 
346         ↪ identifier, '+', '-', '(', '[', 'IF', 'FOR', 'WHILE', 'FUNC'",
347         ↪ currentToken.startPosition));
348
349 }
350
351 protected ParseResult ParseListExpression()
352 {
353     ParseResult result = new ParseResult();
354
355     List<NodeBase> elements = new List<NodeBase>();
356     Position startPosition = currentToken.startPosition.Copy();
357
358     if (currentToken.type != TokenType.LBRACKET)
359         return result.Failure(new InvalidSyntaxError("Expected '['",
360             ↪ currentToken.startPosition));
361
362     result.RegisterAdvancement();
363     Advance();
364
365     if (currentToken.type == TokenType.RBRACKET)
366     {
367         result.RegisterAdvancement();
368         Advance();
369     }
370     else

```



```

366     {
367         elements.Add(result.Register(ParseExpression()));
368         if (result.error != null)
369             return result.Failure(new InvalidSyntaxError("Expected ']', '
                ↳ 'VAR', 'IF', 'FOR', 'WHILE', 'FUNC', number,
                ↳ identifier, '+', '-', '(', '[', or 'NOT'",
                ↳ currentToken.startPosition));

370
371         while (currentToken.type == TokenType.COMMA)
372         {
373             result.RegisterAdvancement();
374             Advance();
375
376             elements.Add(result.Register(ParseExpression()));
377             if (result.error != null) return result;
378         }
379
380         if (currentToken.type != TokenType.RBRACKET)
381             return result.Failure(new InvalidSyntaxError("Expected ']' ",
                ↳ currentToken.startPosition));

382
383         result.RegisterAdvancement();
384         Advance();
385     }
386
387     return result.Success(new ListNode(elements, startPosition,
                ↳ currentToken.endPosition));
388 }
389
390 protected ParseResult ParseSelfExpression()
391 {
392     throw new NotImplementedException();
393 }
394
395 protected ParseResult ParseForExpression()
396 {
397     ParseResult result = new ParseResult();
398
399     if (!currentToken.Matches(TokenType.KEYWORD, "for"))
400         return result.Failure(new InvalidSyntaxError("Expected 'FOR'",
                ↳ currentToken.startPosition));
401
402     result.RegisterAdvancement();
403     Advance();
404
405     if (currentToken.type != TokenType.IDENTIFIER)
406         return result.Failure(new InvalidSyntaxError("Expected
                ↳ identifier", currentToken.startPosition));
407
408     Token varNameToken = currentToken;
409     result.RegisterAdvancement();
410     Advance();
411
412     if (currentToken.type != TokenType.EQUALS)
413         return result.Failure(new InvalidSyntaxError("Expected '=',",
                ↳ currentToken.startPosition));
414
415     result.RegisterAdvancement();
416     Advance();
417
418     NodeBase startValue = result.Register(ParseExpression());
419     if (result.error != null) return result;

```



```

420
421     if (!currentToken.Matches(TokenType.KEYWORD, "to"))
422         return result.Failure(new InvalidSyntaxError("Expected 'TO'",
423             ↪ currentToken.startPosition));
424
425     result.RegisterAdvancement();
426     Advance();
427
428     NodeBase endValue = result.Register(ParseExpression());
429     if (result.error != null) return result;
430
431     NodeBase stepValue = null;
432     if (currentToken.Matches(TokenType.KEYWORD, "step"))
433     {
434         result.RegisterAdvancement();
435         Advance();
436
437         stepValue = result.Register(ParseExpression());
438         if (result.error != null) return result;
439     }
440
441     if (!currentToken.Matches(TokenType.KEYWORD, "do"))
442         return result.Failure(new InvalidSyntaxError("Expected 'DO'",
443             ↪ currentToken.startPosition));
444
445     result.RegisterAdvancement();
446     Advance();
447
448     if (currentToken.type == TokenType.NEWLINE)
449     {
450         result.RegisterAdvancement();
451         Advance();
452
453         NodeBase bodyNodes = result.Register(ParseStatements());
454         if (result.error != null) return result;
455
456         if (!currentToken.Matches(TokenType.KEYWORD, "end"))
457             return result.Failure(new InvalidSyntaxError("Expected 'END'",
458                 ↪ currentToken.startPosition));
459
460         result.RegisterAdvancement();
461         Advance();
462
463         return result.Success(new ForNode(varNameToken, startValue,
464             ↪ endValue, stepValue, bodyNodes, true));
465     }
466
467     NodeBase bodyNode = result.Register(ParseStatement());
468     if (result.error != null) return result;
469
470     return result.Success(new ForNode(varNameToken, startValue,
471         ↪ endValue, stepValue, bodyNode, false));
472 }
473
474 protected ParseResult ParseWhileExpression()
475 {
476     ParseResult result = new ParseResult();
477
478     if (!currentToken.Matches(TokenType.KEYWORD, "while"))
479         return result.Failure(new InvalidSyntaxError("Expected 'WHILE'",
480             ↪ currentToken.startPosition));
481
482     result.RegisterAdvancement();
483     Advance();
484
485     NodeBase bodyNode = result.Register(ParseStatement());
486     if (result.error != null) return result;
487
488     return result.Success(new WhileNode(varNameToken, startValue,
489         ↪ endValue, bodyNode));
490 }

```



```

476     result.RegisterAdvancement();
477     Advance();
478
479     NodeBase condition = result.Register(ParseExpression());
480     if (result.error != null) return result;
481
482     if (!currentToken.Matches(TokenType.KEYWORD, "do"))
483         return result.Failure(new InvalidSyntaxError("Expected 'do'",
484             ↪ currentToken.startPosition));
485
486     result.RegisterAdvancement();
487     Advance();
488
489     NodeBase bodyNode;
490     if (currentToken.type == TokenType.NEWLINE)
491     {
492         result.RegisterAdvancement();
493         Advance();
494
495         bodyNode = result.Register(ParseStatements());
496         if (result.error != null) return result;
497
498         if (!currentToken.Matches(TokenType.KEYWORD, "end"))
499             return result.Failure(new InvalidSyntaxError("Expected 'END'",
500                 ↪ currentToken.startPosition));
501
502         result.RegisterAdvancement();
503         Advance();
504
505         return result.Success(new WhileNode(condition, bodyNode, true));
506     }
507     else
508     {
509         bodyNode = result.Register(ParseStatement());
510         if (result.error != null) return result;
511
512         return result.Success(new WhileNode(condition, bodyNode, false));
513     }
514 }
515
516 protected ParseResult ParseFunctionDefinition()
517 {
518     ParseResult result = new ParseResult();
519
520     if (!currentToken.Matches(TokenType.KEYWORD, "func"))
521         return result.Failure(new InvalidSyntaxError("Expected 'FUNC'",
522             ↪ currentToken.startPosition));
523
524     result.RegisterAdvancement();
525     Advance();
526
527     Token varNameToken = null;
528     if (currentToken.type == TokenType.IDENTIFIER)
529     {
530         varNameToken = currentToken;
531
532         result.RegisterAdvancement();
533         Advance();
534
535         if (currentToken.type != TokenType.LPAREN)
536             return result.Failure(new InvalidSyntaxError("Expected '(',
537                 ↪ currentToken.startPosition));
538     }
539 }

```



```

534     }
535     else
536     {
537         if (currentToken.type != TokenType.LPAREN)
538             return result.Failure(new InvalidSyntaxError("Expected_
                    ↪ identifier_or_('(', currentToken.startPosition));
539     }
540
541     result.RegisterAdvancement();
542     Advance();
543
544     List<Token> argTokens = new List<Token>();
545
546     if (currentToken.type == TokenType.IDENTIFIER)
547     {
548         argTokens.Add(currentToken);
549         result.RegisterAdvancement();
550         Advance();
551
552         while (currentToken.type == TokenType.COMMA)
553         {
554             result.RegisterAdvancement();
555             Advance();
556
557             if (currentToken.type != TokenType.IDENTIFIER)
558                 return result.Failure(new InvalidSyntaxError("Expected_
                    ↪ identifier", currentToken.startPosition));
559
560             argTokens.Add(currentToken);
561             result.RegisterAdvancement();
562             Advance();
563         }
564
565         if (currentToken.type != TokenType.RPAREN)
566             return result.Failure(new InvalidSyntaxError("Expected_'', '_
                    ↪ or_')'", currentToken.startPosition));
567     }
568     else
569     {
570         if (currentToken.type != TokenType.RPAREN)
571             return result.Failure(new InvalidSyntaxError("Expected_
                    ↪ identifier_or_')'", currentToken.startPosition));
572     }
573
574     result.RegisterAdvancement();
575     Advance();
576
577     NodeBase bodyNode;
578     if (currentToken.type == TokenType.ARROW)
579     {
580         result.RegisterAdvancement();
581         Advance();
582
583         bodyNode = result.Register(ParseExpression());
584         if (result.error != null) return result;
585
586         return result.Success(new FunctionDefinitionNode(varNameToken,
                    ↪ argTokens.ToArray(), bodyNode, true));
587     }
588
589     if (currentToken.type != TokenType.NEWLINE)

```



```

590         return result.Failure(new InvalidSyntaxError("Expected '→' or '↪'"
591             ↪ newline", currentToken.startPosition));
592     result.RegisterAdvancement();
593     Advance();
594
595     bodyNode = result.Register(ParseStatements());
596     if (result.error != null) return result;
597
598     if (!currentToken.Matches(TokenType.KEYWORD, "end"))
599         return result.Failure(new InvalidSyntaxError("Expected 'END'",
600             ↪ currentToken.startPosition));
601
602     result.RegisterAdvancement();
603     Advance();
604
605     return result.Success(new FunctionDefinitionNode(varNameToken,
606         ↪ argTokens.ToArray(), bodyNode, false));
607 }
608
609 //#####
610
611 protected delegate ParseResult BinaryOperationDelegate();
612 protected ParseResult ParseBinaryOperation(BinaryOperationDelegate
613     ↪ leftFunc, TokenType[] operations)
614 {
615     return ParseBinaryOperation(leftFunc, operations, leftFunc);
616 }
617
618 protected ParseResult ParseBinaryOperation(BinaryOperationDelegate
619     ↪ leftFunc, TokenType[] operations, BinaryOperationDelegate
620     ↪ rightFunc)
621 {
622     ParseResult result = new ParseResult();
623
624     NodeBase leftNode = result.Register(leftFunc());
625     if (result.error != null)
626         return result;
627
628     while (operations.Contains(currentToken.type))
629     {
630         Token operatorToken = currentToken;
631         result.RegisterAdvancement();
632         Advance();
633
634         NodeBase rightNode = result.Register(rightFunc());
635         if (result.error != null)
636             return result;
637
638         leftNode = new BinaryOperationNode(leftNode, operatorToken,
639             ↪ rightNode);
640     }
641
642     return result.Success(leftNode);
643 }
644
645 protected ParseResult ParseBinaryOperation(BinaryOperationDelegate
646     ↪ leftFunc, (TokenType, string)[] operations)
647 {
648     return ParseBinaryOperation(leftFunc, operations, leftFunc);
649 }
650
651 protected ParseResult ParseBinaryOperation(BinaryOperationDelegate
652     ↪ leftFunc, (TokenType, string)[] operations,
653     ↪ BinaryOperationDelegate rightFunc)

```



```

642     {
643         ParseResult result = new ParseResult();
644
645         NodeBase leftNode = result.Register(leftFunc());
646         if (result.error != null)
647             return result;
648
649         while (operations.Contains((currentToken.type,
650             ↪ (string)currentToken.value)))
651         {
652             Token operatorToken = currentToken;
653             result.RegisterAdvancement();
654             Advance();
655
656             NodeBase rightNode = result.Register(rightFunc());
657             if (result.error != null)
658                 return result;
659
660             leftNode = new BinaryOperationNode(leftNode, operatorToken,
661                 ↪ rightNode);
662         }
663
664         return result.Success(leftNode);
665     }
666 }

```

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();

```



```

37 public Position position { get; protected set; }
38 public string input { get; protected set; }
39 public char currentChar { get; protected set; } = '\0';
40
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
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("Empty Input", "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 }

```



```

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                                 currentChar));
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));
137             Advance();
138             break;
139         case '^':
140             tokens.Add(new Token(TokenType.EXPONENT, position));
141             Advance();
142             break;
143
144         case '!':
145             (Token token, ShorkError error) =
146                 MakeNotEqualsToken();
147             if (error != null) return (null, error);
148             tokens.Add(token);
149             break;
150         case '=':
151             tokens.Add(MakeEqualsToken());
152             break;
153         case '<':
154             tokens.Add(MakeLessThanToken());
155             break;
156         case '>':
157             tokens.Add(MakeGreaterThanToken());
158     }
159 }

```



```

154         break;
155
156     case '.':
157         tokens.Add(new Token(TokenType.DOT, position));
158         Advance();
159         break;
160     case ',':
161         tokens.Add(new Token(TokenType.COMMA, position));
162         Advance();
163         break;
164
165     case '(':
166         tokens.Add(new Token(TokenType.LPAREN, position));
167         Advance();
168         break;
169     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     (Token, ShorkError) MakeStringToken()
220     {
221         Position startPosition = position.Copy();
222         string str = string.Empty;
223         Advance();
224
225         bool escaping = false;
226         while (true)
227         {
228             if (escaping)
229             {
230                 switch (currentChar)
231                 {
232                     default:
233                         return (null, new
234                             ↳ InvalidEscapeSequenceError(string.Format("\\{0}",
235                             ↳ currentChar), position));
236                     case '"':
237                         str += '"';
238                         break;
239                     case '\\':
240                         str += '\\';
241                         break;
242                     case 't':
243                         str += '\t';
244                         break;
245                 }
246                 escaping = false;
247             }
248
249             else if (currentChar == '"')
250             {
251                 Advance();
252                 break;
253             }
254
255             else if (currentChar == '\\')
256                 escaping = true;
257
258             else
259                 str += currentChar;
260
261             Advance();
262         }
263
264         return (new Token(TokenType.STRING, str, startPosition, position),
265             ↳ null);
266     }
267
268     Token MakeIdentifierToken()
269     {
270         Position startPosition = position.Copy();
271         string idstr = string.Empty + currentChar;
272         Advance();
273
274         while (LETTERS_WITH_UNDERSCORE.Contains(currentChar))
275         {
276             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;
                return new Token(ttype, idstr, startPosition, position);
286     }
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

```

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}',_{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
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 bool Matches(params TokenType[] types)
43     {
44         foreach (TokenType tpe in types)
45             if (this.type == tpe) return true;
46         return this.type == types[0];
47     }
48     public bool Matches(TokenType type, dynamic value)
49     {
50         if (type == TokenType.KEYWORD)
51             return this.type == type && ((string)this.value).ToLower() ==
52                 ↪ ((string)value).ToLower();
53         return this.type == type && this.value == value;
54     }
55
56     public override string ToString()
57     {
58         if (value == null)
59             return string.Format("[{0}]", type);
60         else
61             return string.Format("[{0}]:{1}", type, value);
62     }
63 }

```


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 }
```