Shork#

Miss Ylva Llywelyn

2023/10/14

Based on the tutorial series by David Callanan.

Contents

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.

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

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
               ↪ startPosition, Position endPosition)
20             : base(startPosition, endPosition)
21         {
22             this.statements = statements.ToList();
23         }
24
25         public override string ToString()
26         {
27             return string.Format("{{{0}}}", string.Join(",␣", statements));
28         }
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          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)
68              : base(startPosition, endPosition)
69          {
70              this.elementNodes = elementNodes.ToList();
71          }
72
73          public override string ToString()
74          {
75              return string.Format("[{0}]", string.Join(",␣", elementNodes));
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,
                ↪ NodeBase rightNode)
120             : base(leftNode.startPosition, rightNode.endPosition)
121         {
122             this.leftNode = leftNode;
123             this.operatorToken = operatorToken;
124             this.rightNode = rightNode;
125         }
126 
127         public override string ToString()
128         {
129             return string.Format("({0} {1} {2})", leftNode, operatorToken,
                    ↪ rightNode);
130         }
131     }
132 
133     public class UnaryOperationNode : NodeBase
134     {
135         public Token operatorToken { get; protected set; }
136         public NodeBase operandNode { get; protected set; }
137 
138         public UnaryOperationNode(Token operatorToken, NodeBase operandNode)
139             : base(operatorToken.startPosition, operandNode.endPosition)
140         {
141             this.operatorToken = operatorToken;
142             this.operandNode = operandNode;
143         }
144     }
145 
146     public class IfNode : NodeBase
147     {
148         public (NodeBase, NodeBase)[] caseNodes { get; protected set; }
149         public NodeBase elseNode { get; protected set; }
150 
151         public IfNode((NodeBase, NodeBase)[] caseNodes)
152             : base(caseNodes[0].Item1.startPosition,
                    ↪ caseNodes[^1].Item2.endPosition)
153         {
154             this.caseNodes = caseNodes;
155         }
156         public IfNode((NodeBase, NodeBase)[] caseNodes, NodeBase elseNode)
157             : base(caseNodes[0].Item1.startPosition, elseNode.endPosition)
158         {
159             this.caseNodes = caseNodes;
160             this.elseNode = elseNode;
161         }
162     }
163 
164     public class ForNode : NodeBase
165     {
166         public Token varNameToken { get; protected set; }
167         public NodeBase startValueNode { get; protected set; }
168         public NodeBase endValueNode { get; protected set; }
169         public NodeBase stepValueNode { get; protected set; }
170         public NodeBase bodyNode { get; protected set; }
171         public bool shouldReturnNull { get; protected set; }
172 
173         public ForNode(Token varNameToken,
174                        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
                ↪ shouldReturnNull)
197             : base(conditionNode.startPosition, bodyNode.endPosition)
198         {
199             this.conditionNode = conditionNode;
200             this.bodyNode = bodyNode;
201             this.shouldReturnNull = shouldReturnNull;
202         }
203     }
204
205     public class FunctionDefinitionNode : NodeBase
206     {
207         public Token varNameToken { get; protected set; }
208         public Token[] argNameTokens { get; protected set; }
209         public NodeBase bodyNode { get; protected set; }
210         public bool shouldAutoReturn { get; protected set; }
211
212         public FunctionDefinitionNode(Token varNameToken,
213                                       Token[] argNameTokens,
214                                       NodeBase bodyNode,
215                                       bool shouldAutoReturn)
216             : base(varNameToken.startPosition, bodyNode.endPosition)
217         {
218             this.varNameToken = varNameToken;
219             this.argNameTokens = argNameTokens;
220             this.bodyNode = bodyNode;
221             this.shouldAutoReturn = shouldAutoReturn;
222         }
223     }
224
225     public class CallNode : NodeBase
226     {
227         public NodeBase nodeToCall { get; protected set; }
228         public NodeBase[] argumentNodes { get; protected set; }
229
230         public CallNode(NodeBase nodeToCall, NodeBase[] argumentNodes)
231             : base(nodeToCall.startPosition, (argumentNodes.Length > 0) ?
                    ↪ argumentNodes[^1].endPosition : nodeToCall.endPosition)
232         {
233             this.nodeToCall = nodeToCall;
234             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) ?
                   ↪ this.tokens[tokenIndex] : null;
19             return currentToken;
20         }
21
22         Token Reverse(int amount = 1)
23         {
24             tokenIndex -= amount;
25             currentToken = (tokenIndex < tokens.Length) ?
                   ↪ this.tokens[tokenIndex] : null;
26             return currentToken;
27         }
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",
                        ↪ currentToken.startPosition));
35
36              return result;
37          }
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,
                    ↪ currentToken.endPosition));
85          }
86
87          protected ParseResult ParseStatement()
88          {
89              ParseResult result = new ParseResult();
90              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,
                            ↪ currentToken.endPosition));
102                 }
103                 else
104                     return result.Success(new ReturnNode(expression));
105             }
106
107             else if (currentToken.Matches(TokenType.KEYWORD, "continue"))
108             {
109                 result.RegisterAdvancement();
110                 Advance();
111                 return result.Success(new ContinueNode(startPosition,
                        ↪ currentToken.endPosition));
112             }
113
114             else if (currentToken.Matches(TokenType.KEYWORD, "break"))
115             {
116                 result.RegisterAdvancement();
117                 Advance();
118                 return result.Success(new BreakNode(startPosition,
                        ↪ currentToken.endPosition));
119             }
120
121             else
122             {
123                 NodeBase expression = result.Register(ParseExpression());
124                 if (result.error != null)
125                     return result.Failure(new InvalidSyntaxError("Expected␣
                            ↪ 'RETURN',␣'CONTINUE',␣'BREAK',␣'VAR',␣'IF',␣'FOR',␣
                            ↪ 'WHILE',␣'FUN',␣int,␣float,␣identifier,␣'+',␣'-',␣'(',␣
                            ↪ '['␣or␣'NOT'", currentToken.startPosition));
126
127                 return result.Success(expression);
128             }
129         }
130
131         protected ParseResult ParseExpression()
132         {
133             ParseResult result = new ParseResult();
134
135             if (currentToken.Matches(TokenType.KEYWORD, "var"))
136             {
137                 result.RegisterAdvancement();
138                 Advance();
139
140                 if (currentToken.type != TokenType.IDENTIFIER)
141                     return result.Failure(new InvalidSyntaxError("Expected␣
                            ↪ identifier", currentToken.startPosition));
142
143                 Token varNameToken = currentToken;
144                 result.RegisterAdvancement();
145                 Advance();
```

```
146
147                 if (currentToken.type != TokenType.EQUALS)
148                     return result.Failure(new InvalidSyntaxError("Expected '='",
                        ↪ 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,
                        ↪ expression));
156             }
157
158             else
159             {
160                 NodeBase node =
                        ↪ result.Register(ParseBinaryOperation(ParseComparisonExpression,
                        ↪ new (TokenType, string)[] { (TokenType.KEYWORD, "and"),
                        ↪ (TokenType.KEYWORD, "or") }));
161                 if (result.error != null)
162                     return result.Failure(new InvalidSyntaxError("Expected
                            ↪ 'VAR', 'IF', 'FOR', 'WHILE', 'FUNC', number,
                            ↪ identifier, '+', '-', '(', '[' or 'NOT'",
                            ↪ currentToken.startPosition));
163                 return result.Success(node);
164             }
165         }
166
167         protected ParseResult ParseComparisonExpression()
168         {
169             ParseResult result = new ParseResult();
170             NodeBase node;
171
172             if (currentToken.Matches(TokenType.KEYWORD, "not"))
173             {
174                 Token operatorToken = currentToken;
175                 result.RegisterAdvancement();
176                 Advance();
177
178                 node = result.Register(ParseComparisonExpression());
179                 if (result.error != null) return result;
180                 return result.Success(node);
181             }
182
183             node =
                    ↪ result.Register(ParseBinaryOperation(ParseArithmaticExpression,
                    ↪ new TokenType[] { TokenType.DOUBLE_EQUALS,
                    ↪ TokenType.NOT_EQUALS, TokenType.LESS_THAN,
                    ↪ TokenType.GREATER_THAN, TokenType.LESS_THAN_OR_EQUAL,
                    ↪ TokenType.GREATER_THAN_OR_EQUAL }));
184             if (result.error != null)
185                 return result.Failure(new InvalidSyntaxError("Expected number,
                        ↪ identifier, '+', '-', '(', '[', 'IF', 'FOR', 'WHILE',
                        ↪ 'FUNC' or 'NOT'", currentToken.startPosition));
186             return result.Success(node);
187         }
188
189         protected ParseResult ParseArithmaticExpression()
190         {
191             return ParseBinaryOperation(ParseTerm, new TokenType[] {
                    ↪ TokenType.PLUS, TokenType.MINUS });
```

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

```
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 =
                        ↪ result.Register(ParseFunctionDefinition());
340                 if (result.error != null) return result;
341                 return result.Success(functionDefinition);
342             }
343
344             else return result.Failure(new InvalidSyntaxError("Expected␣number,␣
                    ↪ identifier,␣'+',␣'-',␣'(',␣'[',␣IF',␣'FOR',␣'WHILE',␣'FUNC'",
                    ↪ currentToken.startPosition));
345         }
346
347         protected ParseResult ParseListExpression()
348         {
349             ParseResult result = new ParseResult();
350
351             List<NodeBase> elements = new List<NodeBase>();
352             Position startPosition = currentToken.startPosition.Copy();
353
354             if (currentToken.type != TokenType.LBRACKET)
355                 return result.Failure(new InvalidSyntaxError("Expected␣'['",
                        ↪ currentToken.startPosition));
356
357             result.RegisterAdvancement();
358             Advance();
359
360             if (currentToken.type == TokenType.RBRACKET)
361             {
362                 result.RegisterAdvancement();
363                 Advance();
364             }
365             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         // TODO: ParseIfExpression
391         protected ParseResult ParseIfExpression()
392         {
393             throw new NotImplementedException();
394         }
395 
396         protected ParseResult ParseForExpression()
397         {
398             ParseResult result = new ParseResult();
399 
400             if (!currentToken.Matches(TokenType.KEYWORD, "for"))
401                 return result.Failure(new InvalidSyntaxError("Expected 'FOR'",
                    ↪ currentToken.startPosition));
402 
403             result.RegisterAdvancement();
404             Advance();
405 
406             if (currentToken.type != TokenType.IDENTIFIER)
407                 return result.Failure(new InvalidSyntaxError("Expected 
                    ↪ identifier", currentToken.startPosition));
408 
409             Token varNameToken = currentToken;
410             result.RegisterAdvancement();
411             Advance();
412 
413             if (currentToken.type != TokenType.EQUALS)
414                 return result.Failure(new InvalidSyntaxError("Expected '='",
                    ↪ currentToken.startPosition));
415 
416             result.RegisterAdvancement();
417             Advance();
418 
419             NodeBase startValue = result.Register(ParseExpression());
```

```
420             if (result.error != null) return result;
421
422             if (!currentToken.Matches(TokenType.KEYWORD, "to"))
423                 return result.Failure(new InvalidSyntaxError("Expected 'TO'",
                        ↪ 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'",
                        ↪ currentToken.startPosition));
443
444             result.RegisterAdvancement();
445             Advance();
446
447             if (currentToken.type == TokenType.NEWLINE)
448             {
449                 result.RegisterAdvancement();
450                 Advance();
451
452                 NodeBase bodyNodes = result.Register(ParseStatements());
453                 if (result.error != null) return result;
454
455                 if (!currentToken.Matches(TokenType.KEYWORD, "end"))
456                     return result.Failure(new InvalidSyntaxError("Expected
                            ↪ 'END'", currentToken.startPosition));
457
458                 result.RegisterAdvancement();
459                 Advance();
460
461                 return result.Success(new ForNode(varNameToken, startValue,
                        ↪ endValue, stepValue, bodyNodes, true));
462             }
463
464             NodeBase bodyNode = result.Register(ParseStatement());
465             if (result.error != null) return result;
466
467             return result.Success(new ForNode(varNameToken, startValue,
                    ↪ endValue, stepValue, bodyNode, false));
468         }
469
470         protected ParseResult ParseWhileExpression()
471         {
472             throw new NotImplementedException();
473         }
474
475         protected ParseResult ParseFunctionDefinition()
476         {
```

```
477             throw new NotImplementedException();
478         }
479 
480         //#####################################
481 
482         protected delegate ParseResult BinaryOperationDelegate();
483         protected ParseResult ParseBinaryOperation(BinaryOperationDelegate
                ↪ leftFunc, TokenType[] operations)
484         {
485             return ParseBinaryOperation(leftFunc, operations, leftFunc);
486         }
487         protected ParseResult ParseBinaryOperation(BinaryOperationDelegate
                ↪ leftFunc, TokenType[] operations, BinaryOperationDelegate
                ↪ rightFunc)
488         {
489             ParseResult result = new ParseResult();
490 
491             NodeBase leftNode = result.Register(leftFunc());
492             if (result.error != null)
493                 return result;
494 
495             while (operations.Contains(currentToken.type))
496             {
497                 Token operatorToken = currentToken;
498                 result.RegisterAdvancement();
499                 Advance();
500 
501                 NodeBase rightNode = result.Register(rightFunc());
502                 if (result.error != null)
503                     return result;
504 
505                 leftNode = new BinaryOperationNode(leftNode, operatorToken,
                        ↪ rightNode);
506             }
507 
508             return result.Success(leftNode);
509         }
510         protected ParseResult ParseBinaryOperation(BinaryOperationDelegate
                ↪ leftFunc, (TokenType, string)[] operations)
511         {
512             return ParseBinaryOperation(leftFunc, operations, leftFunc);
513         }
514         protected ParseResult ParseBinaryOperation(BinaryOperationDelegate
                ↪ leftFunc, (TokenType, string)[] operations,
                ↪ BinaryOperationDelegate rightFunc)
515         {
516             ParseResult result = new ParseResult();
517 
518             NodeBase leftNode = result.Register(leftFunc());
519             if (result.error != null)
520                 return result;
521 
522             while (operations.Contains((currentToken.type,
                    ↪ (string)currentToken.value)))
523             {
524                 Token operatorToken = currentToken;
525                 result.RegisterAdvancement();
526                 Advance();
527 
528                 NodeBase rightNode = result.Register(rightFunc());
529                 if (result.error != null)
530                     return result;
```

```
531
532                 leftNode = new BinaryOperationNode(leftNode, operatorToken,
                        ↪ rightNode);
533             }
534
535             return result.Success(leftNode);
536         }
537     }
538 }
```

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
           ↪ tokens.
5      /// </summary>
6      public class Lexer
7      {
8          /// <summary>
9          /// The words recognised as keywords.
10         /// </summary>
11         static readonly string[] KEYWORDS =
12         {
13             "var",
14             "and",
15             "or",
16             "not",
17             "if",
18             "then",
19             "elif",
20             "else",
21             "for",
22             "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 occured, 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("Empty␣Input", "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}'",
```

```
                                    ↪ currentChar), position));
111                         case '+':
112                             tokens.Add(new Token(TokenType.PLUS, position));
113                             Advance();
114                             break;
115                         case '-':
116                             TokenType ttype = TokenType.MINUS;
117                             Position startPosition = position.Copy();
118                             Advance();
119
120                             if (currentChar == '>')
121                             {
122                                 ttype = TokenType.ARROW;
123                                 Advance();
124                             }
125
126                             tokens.Add(new Token(ttype, startPosition,
                                    ↪ position));
127                             break;
128                         case '*':
129                             tokens.Add(new Token(TokenType.MULTIPLY, position));
130                             Advance();
131                             break;
132                         case '/':
133                             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) =
                                    ↪ MakeNotEqualsToken();
143                             if (error != null) return (null, error);
144                             tokens.Add(token);
145                             break;
146                         case '=':
147                             tokens.Add(MakeEqualsToken());
148                             break;
149                         case '<':
150                             tokens.Add(MakeLessThanToken());
151                             break;
152                         case '>':
153                             tokens.Add(MakeGreaterThanToken());
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),
                ↪ startPosition, position);
217         }
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
                                        ↪ 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(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

```
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
                ↪ 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,
                ↪ Position endPosition)
33          {
34              this.type = type;
35              this.value = value;
36              this.startPosition = startPosition.Copy();
37              this.endPosition = endPosition.Copy();
38          }
39
40          public bool Matches(params TokenType[] types)
41          {
42              foreach (TokenType ttpe in types)
43                  if (this.type == type) return true;
44              return this.type == type;
45          }
46          public bool Matches(TokenType type, dynamic value)
47          {
48              if (type == TokenType.KEYWORD)
49                  return this.type == type && ((string)this.value).ToLower() ==
                        ↪ ((string)value).ToLower();
50              return this.type == type && this.value == value;
51          }
52
53          public override string ToString()
54          {
55              if (value == null)
56                  return string.Format("[{0}]", type);
57              else
58                  return string.Format("[{0} : {1}]", type, value);
59          }
60      }
61  }
```

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  }
```