Shork#

Miss Ylva Llywelyn

2023/10/14

Contents

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:VAR IDENTIFIER = *expression* KEYWORD:CONTINUE KEYWORD:BREAK *expression*
expression	

Code Listing

Listing 1: NodeBase.cs

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

```
59              return string.Format("(List{{{0}}})", string.Join(",␣",
                    ↪ elementNodes));
60          }
61      }
62
63      public class VarAssignNode : NodeBase
64      {
65          public Token varNameToken { get; protected set; }
66          public NodeBase valueNode { get; protected set; }
67
68          public VarAssignNode(Token varNameToken, NodeBase valueNode)
69              : base(varNameToken.startPosition, valueNode.endPosition)
70          {
71              this.varNameToken = varNameToken;
72              this.valueNode = valueNode;
73          }
74
75          public override string ToString()
76          {
77              return string.Format("({0}␣=␣{1})", varNameToken, valueNode);
78          }
79      }
80
81      public class VarAccessNode : NodeBase
82      {
83          public Token varNameToken { get; protected set; }
84
85          public VarAccessNode(Token varNameToken)
86              : base(varNameToken.startPosition, varNameToken.endPosition)
87          {
88              this.varNameToken = varNameToken;
89          }
90
91          public override string ToString()
92          {
93              return string.Format("({0})", varNameToken);
94          }
95      }
96
97      public class BinaryOperationNode : NodeBase
98      {
99          public NodeBase leftNode { get; protected set; }
100         public Token operatorToken { get; protected set; }
101         public NodeBase rightNode { get; protected set; }
102
103         public BinaryOperationNode(NodeBase leftNode, Token operatorToken,
                ↪ NodeBase rightNode)
104             : base(leftNode.startPosition, rightNode.endPosition)
105         {
106             this.leftNode = leftNode;
107             this.operatorToken = operatorToken;
108             this.rightNode = rightNode;
109         }
110
111         public override string ToString()
112         {
113             return string.Format("({0}␣{1}␣{2})", leftNode, operatorToken,
                    ↪ rightNode);
114         }
115     }
116
117     public class UnaryOperationNode : NodeBase
```

```
118     {
119         public Token operatorToken { get; protected set; }
120         public NodeBase operandNode { get; protected set; }
121 
122         public UnaryOperationNode(Token operatorToken, NodeBase operandNode)
123             : base(operatorToken.startPosition, operandNode.endPosition)
124         {
125             this.operatorToken = operatorToken;
126             this.operandNode = operandNode;
127         }
128     }
129 }
```

Listing 2: Parser.cs

```
1  namespace ShorkSharp
2  {
3      public class Parser
4      {
5          Token[] tokens;
6          int tokenIndex = 0;
7          Token currentToken;
8 
9          public Parser(Token[] tokens)
10         {
11             this.tokens = tokens;
12             this.currentToken = this.tokens[0];
13         }
14 
15         public void Advance()
16         {
17             tokenIndex++;
18             currentToken = (tokenIndex < tokens.Length) ?
                   ↪ this.tokens[tokenIndex] : null;
19         }
20 
21         public ParseResult Parse()
22         {
23             ParseResult result = ParseExpression();
24 
25             if (result.error != null && currentToken.type != TokenType.EOF)
26                 return result.Failure(new InvalidSyntaxError("Unexpected␣EOF",
                       ↪ currentToken.startPosition));
27 
28             return result;
29         }
30 
31         //###################################
32 
33         protected ParseResult ParseExpression()
34         {
35             throw new NotImplementedException();
36         }
37     }
38 }
```

Listing 3: ParseResult.cs

```
1  namespace ShorkSharp
2  {
3      public class ParseResult
4      {
5          public ShorkError error { get; protected set; }
6          public NodeBase node { get; protected set; }
```

```
7           public int advanceCount { get; protected set; } = 0;
8
9           public ParseResult() { }
10
11          public void RegisterAdvancement()
12          {
13              advanceCount++;
14          }
15
16          public NodeBase Register(ParseResult result)
17          {
18              this.advanceCount += result.advanceCount;
19              if (result.error != null) this.error = result.error;
20              return result.node;
21          }
22
23          public ParseResult Success(NodeBase node)
24          {
25              this.node = node;
26              return this;
27          }
28
29          public ParseResult Failure(ShorkError error)
30          {
31              if (this.error == null || this.advanceCount == 0)
32                  this.error = error;
33              return this;
34          }
35      }
36  }
```

Listing 4: Lexer.cs

```
1   namespace ShorkSharp
2   {
3       public class Lexer
4       {
5           static readonly string[] KEYWORDS =
6           {
7               "var",
8               "and",
9               "or",
10              "not",
11              "if",
12              "then",
13              "elif",
14              "else",
15              "for",
16              "to",
17              "step",
18              "func",
19              "while",
20              "do",
21              "end",
22              "return",
23              "continue",
24              "break"
25          };
26          static readonly char[] WHITESPACE = { ' ', '\t', '\r' };
27          static readonly char[] DIGITS = { '0', '1', '2', '3', '4', '5', '6',
                ↪ '7', '8', '9' };
28          static readonly char[] DIGITS_WITH_DOT = DIGITS.Concat(new char[] { '.'
                ↪ }).ToArray();
```

```
29          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' };
30          static readonly char[] LETTERS_WITH_UNDERSCORE = LETTERS.Concat(new
                ↪ char[] { '_' }).ToArray();
31
32          public Position position { get; protected set; }
33          public string input { get; protected set; }
34          public char currentChar { get; protected set; } = '\0';
35
36          public Lexer(string input)
37          {
38              this.input = input;
39              this.position = new Position(input);
40          }
41          public Lexer(string input, string filename)
42          {
43              this.input = input;
44              this.position = new Position(filename);
45          }
46
47          void Advance()
48          {
49              position.Advance(currentChar);
50
51              if (position.index < input.Length)
52                  currentChar = input[position.index];
53              else
54                  currentChar = '\0';
55          }
56
57          public (Token[], ShorkError?) Lex()
58          {
59              if (input.Length == 0)
60                  return (new Token[] { }, new ShorkError("Empty␣Input", "Input␣
                        ↪ text␣is␣empty", null));
61              this.currentChar = input[0];
62
63              List<Token> tokens = new List<Token>();
64
65              while (currentChar != '\0')
66              {
67                  if (WHITESPACE.Contains(currentChar))
68                  {
69                      Advance();
70                  }
71
72                  // Number Tokens
73                  else if (DIGITS.Contains(currentChar))
74                  {
75                      tokens.Add(MakeNumberToken());
76                  }
77
78                  // String Tokens
79                  else if (currentChar == '"')
80                  {
81                      (Token token, ShorkError error) = MakeStringToken();
82                      if (error != null)
83                          return (null, error);
84                      tokens.Add(token);
```

```
85                  }
86
87                  // Identifiers and Keywords
88                  else if (LETTERS.Contains(currentChar))
89                  {
90                      tokens.Add(MakeIdentifierToken());
91                  }
92
93                  // Simple tokens
94                  else
95                  {
96                      switch (currentChar)
97                      {
98                          default:
99                              return (new Token[] { },
100                                     new
                                           ↪ InvalidCharacterError(string.Format("'{0}'",
                                           ↪ currentChar), position));
101                         case '+':
102                             tokens.Add(new Token(TokenType.PLUS, position));
103                             Advance();
104                             break;
105                         case '-':
106                             TokenType ttype = TokenType.MINUS;
107                             Position startPosition = position.Copy();
108                             Advance();
109
110                             if (currentChar == '>')
111                             {
112                                 ttype = TokenType.ARROW;
113                                 Advance();
114                             }
115
116                             tokens.Add(new Token(ttype, startPosition,
                                    ↪ position));
117                             break;
118                         case '*':
119                             tokens.Add(new Token(TokenType.MULTIPLY, position));
120                             Advance();
121                             break;
122                         case '/':
123                             tokens.Add(new Token(TokenType.DIVIDE, position));
124                             Advance();
125                             break;
126                         case '^':
127                             tokens.Add(new Token(TokenType.EXPONENT, position));
128                             Advance();
129                             break;
130
131                         case '!':
132                             (Token token, ShorkError error) =
                                    ↪ MakeNotEqualsToken();
133                             if (error != null) return (null, error);
134                             tokens.Add(token);
135                             break;
136                         case '=':
137                             tokens.Add(MakeEqualsToken());
138                             break;
139                         case '<':
140                             tokens.Add(MakeLessThanToken());
141                             break;
142                         case '>':
```

```
143                             tokens.Add(MakeGreaterThanToken());
144                             break;
145
146                         case '.':
147                             tokens.Add(new Token(TokenType.DOT, position));
148                             Advance();
149                             break;
150                         case ',':
151                             tokens.Add(new Token(TokenType.COMMA, position));
152                             Advance();
153                             break;
154
155                         case '(':
156                             tokens.Add(new Token(TokenType.LPAREN, position));
157                             Advance();
158                             break;
159                         case ')':
160                             tokens.Add(new Token(TokenType.RPAREN, position));
161                             Advance();
162                             break;
163                         case '{':
164                             tokens.Add(new Token(TokenType.LBRACE, position));
165                             Advance();
166                             break;
167                         case '}':
168                             tokens.Add(new Token(TokenType.RBRACE, position));
169                             Advance();
170                             break;
171                         case '[':
172                             tokens.Add(new Token(TokenType.LBRACKET, position));
173                             Advance();
174                             break;
175                         case ']':
176                             tokens.Add(new Token(TokenType.RBRACKET, position));
177                             Advance();
178                             break;
179                     }
180                 }
181             }
182
183             return (tokens.ToArray(), null);
184         }
185
186         Token MakeNumberToken()
187         {
188             string numstring = string.Empty + currentChar;
189             bool hasDecimalPoint = false;
190             Position startPosition = position.Copy();
191
192             Advance();
193             while (DIGITS_WITH_DOT.Contains(currentChar))
194             {
195                 if (currentChar == '.')
196                 {
197                     if (hasDecimalPoint)
198                         break;
199                     else
200                         hasDecimalPoint = true;
201                 }
202                 numstring += currentChar;
203                 Advance();
204             }
```

```
205
206             return new Token(TokenType.NUMBER, decimal.Parse(numstring),
                    ↪ startPosition, position);
207         }
208
209         (Token, ShorkError) MakeStringToken()
210         {
211             Position startPosition = position.Copy();
212             string str = string.Empty;
213             Advance();
214
215             bool escaping = false;
216             while (true)
217             {
218                 if (escaping)
219                 {
220                     switch (currentChar)
221                     {
222                         default:
223                             return (null, new
                                    ↪ InvalidEscapeSequenceError(string.Format("\\{0}",
                                    ↪ currentChar), position));
224                         case '"':
225                             str += '"';
226                             break;
227                         case '\\':
228                             str += '\\';
229                             break;
230                         case 't':
231                             str += '\t';
232                             break;
233                     }
234                     escaping = false;
235                 }
236
237                 else if (currentChar == '"')
238                 {
239                     Advance();
240                     break;
241                 }
242
243                 else if (currentChar == '\\')
244                     escaping = true;
245
246                 else
247                     str += currentChar;
248
249                 Advance();
250             }
251
252             return (new Token(TokenType.STRING, str, startPosition, position),
                    ↪ null);
253         }
254
255         Token MakeIdentifierToken()
256         {
257             Position startPosition = position.Copy();
258             string idstr = string.Empty + currentChar;
259             Advance();
260
261             while (LETTERS_WITH_UNDERSCORE.Contains(currentChar))
262             {
```

```
263                 idstr += currentChar;
264                 Advance();
265             }
266 
267             if (idstr == "true")
268                 return new Token(TokenType.BOOL, true, startPosition, position);
269             else if (idstr == "false")
270                 return new Token(TokenType.BOOL, false, startPosition, position);
271             else if (idstr == "null")
272                 return new Token(TokenType.NULL, startPosition, position);
273             else
274             {
275                 TokenType ttype = KEYWORDS.Contains(idstr.ToLower()) ?
                        ↪ TokenType.KEYWORD : TokenType.IDENTIFIER;
276                 return new Token(ttype, idstr, startPosition, position);
277             }
278         }
279 
280         Token MakeEqualsToken()
281         {
282             Position startPosition = position.Copy();
283             TokenType ttype = TokenType.EQUALS;
284             Advance();
285             if (currentChar == '=')
286             {
287                 ttype = TokenType.DOUBLE_EQUALS;
288                 Advance();
289             }
290             return new Token(ttype, startPosition, position);
291         }
292 
293         (Token, ShorkError) MakeNotEqualsToken()
294         {
295             Position startPosition = position.Copy();
296             Advance();
297             if (currentChar == '=')
298             {
299                 Advance();
300                 return (new Token(TokenType.NOT_EQUALS, startPosition,
                        ↪ position), null);
301             }
302             return (null, new InvalidCharacterError("", position));
303         }
304 
305         Token MakeLessThanToken()
306         {
307             Position startPosition = position.Copy();
308             TokenType ttype = TokenType.LESS_THAN;
309             Advance();
310             if (currentChar == '=')
311             {
312                 ttype = TokenType.LESS_THAN_OR_EQUAL;
313                 Advance();
314             }
315             return new Token(ttype, startPosition, position);
316         }
317 
318         Token MakeGreaterThanToken()
319         {
320             Position startPosition = position.Copy();
321             TokenType ttype = TokenType.GREATER_THAN;
322             Advance();
```

```
323                 if (currentChar == '=')
324                 {
325                     ttype = TokenType.GREATER_THAN_OR_EQUAL;
326                     Advance();
327                 }
328                 return new Token(ttype, startPosition, position);
329             }
330         }
331     }
```

Listing 5: ShorkError.cs

```
1   using System.Text;
2
3   namespace ShorkSharp
4   {
5       public class ShorkError
6       {
7           public string errorName { get; protected set; }
8           public string details { get; protected set; }
9
10          public Position startPosition { get; protected set; }
11
12          public ShorkError(string errorName, string details, Position
                ↪ startPosition)
13          {
14              this.errorName = errorName;
15              this.details = details;
16              this.startPosition = startPosition;
17          }
18
19          public override string ToString()
20          {
21              StringBuilder sb = new StringBuilder();
22
23              sb.AppendFormat("{0}: {1}", errorName, details);
24
25              if (startPosition != null)
26                  sb.AppendFormat("\nFile: '{0}', line {1}",
                        ↪ startPosition.filename, startPosition.line+1);
27
28              return sb.ToString();
29          }
30      }
31
32      public class InvalidCharacterError : ShorkError
33      {
34          public InvalidCharacterError(string details, Position startPosition)
35              : base("Invalid Character", details, startPosition) { }
36      }
37
38      public class InvalidSyntaxError : ShorkError
39      {
40          public InvalidSyntaxError(string details, Position startPosition)
41              : base("Invalid Syntax", details, startPosition) { }
42      }
43
44      public class InvalidEscapeSequenceError : ShorkError
45      {
46          public InvalidEscapeSequenceError(string details, Position startPosition)
47              : base("Invalid Escape Sequence", details, startPosition) { }
48      }
49  }
```

Listing 6: Token.cs

```
1  namespace ShorkSharp
2  {
3      public class Token
4      {
5          public TokenType type { get; protected set; }
6          public dynamic value { get; protected set; }
7
8          public Position startPosition { get; protected set; }
9          public Position endPosition { get; protected set; }
10
11         public Token(TokenType type, Position startPosition)
12         {
13             this.type = type;
14             this.value = null;
15             this.startPosition = startPosition.Copy();
16             this.endPosition = startPosition.Copy();
17         }
18         public Token(TokenType type, Position startPosition, Position
               ↪ 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 override string ToString()
41         {
42             if (value == null)
43                 return string.Format("[{0}]", type);
44             else
45                 return string.Format("[{0} : {1}]", type, value);
46         }
47     }
48 }
```

Listing 7: TokenType.cs

```
1  namespace ShorkSharp
2  {
3      public enum TokenType
4      {
5          NUMBER,
6          STRING,
7          BOOL,
8          NULL,
9
```

```
10              KEYWORD,
11              IDENTIFIER,
12
13              PLUS,
14              MINUS,
15              MULTIPLY,
16              DIVIDE,
17              EXPONENT,
18
19              EQUALS,
20              DOUBLE_EQUALS,
21              NOT_EQUALS,
22              LESS_THAN,
23              GREATER_THAN,
24              LESS_THAN_OR_EQUAL,
25              GREATER_THAN_OR_EQUAL,
26
27              DOT,
28              COMMA,
29              ARROW,
30
31              LPAREN,
32              RPAREN,
33              LBRACE,
34              RBRACE,
35              LBRACKET,
36              RBRACKET,
37
38              NEWLINE,
39              EOF
40      }
41  }
```