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: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*)*

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       /// <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 }
```

Listing 5: ShorkError.cs

```
1  namespace ShorkSharp
2  {
3      public class ShorkError
4      {
5          public string errorName { get; protected set; }
6          public string details { get; protected set; }
7 
8          public Position startPosition { get; protected set; }
9 
10         public ShorkError(string errorName, string details, Position
               ↪ 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  }
```

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 }
```