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

```
namespace ShorkSharp
{
    public abstract class NodeBase
    {
        public Position startPosition { get; protected set; }
        public Position endPosition { get; protected set; }

        protected NodeBase(Position startPosition, Position endPosition)
        {
            this.startPosition = startPosition.Copy();
            this.endPosition = endPosition.Copy();
        }
    }

    public class CodeBlockNode : NodeBase
    {
        public List<NodeBase> statements;

        public CodeBlockNode(IEnumerable<NodeBase> statements, Position
            ↪ startPosition, Position endPosition)
            : base(startPosition, endPosition)
        {
            this.statements = statements.ToList();
        }

        public override string ToString()
        {
            return string.Format("{{{0}}}", string.Join(",␣", statements));
        }
    }

    public class NumberNode : NodeBase
    {
        public Token numToken { get; protected set; }

        public NumberNode(Token numToken)
            : base(numToken.startPosition, numToken.endPosition)
        {
            this.numToken = numToken;
        }

        public override string ToString()
        {
            return string.Format("({0})", numToken);
        }
    }

    public class StringNode : NodeBase
    {
        public Token strToken { get; protected set; }

        public StringNode(Token strToken)
            : base(strToken.startPosition, strToken.endPosition)
        {
            this.strToken = strToken;
        }

        public override string ToString()
        {
            return string.Format("({0})", strToken);
        }
    }

    public class ListNode : NodeBase
    {
        public List<NodeBase> elementNodes;

        public ListNode(IEnumerable<NodeBase> elementNodes, Position
          ↪ startPosition, Position endPosition)
            : base(startPosition, endPosition)
        {
            this.elementNodes = elementNodes.ToList();
        }

        public override string ToString()
        {
            return string.Format("[{0}]", string.Join(",␣", elementNodes));
        }
    }

    public class VarAssignNode : NodeBase
    {
        public Token varNameToken { get; protected set; }
        public NodeBase valueNode { get; protected set; }

        public VarAssignNode(Token varNameToken, NodeBase valueNode)
            : base(varNameToken.startPosition, valueNode.endPosition)
        {
            this.varNameToken = varNameToken;
            this.valueNode = valueNode;
        }

        public override string ToString()
        {
            return string.Format("({0}␣=␣{1})", varNameToken, valueNode);
        }
    }

    public class VarAccessNode : NodeBase
    {
        public Token varNameToken { get; protected set; }

        public VarAccessNode(Token varNameToken)
            : base(varNameToken.startPosition, varNameToken.endPosition)
        {
            this.varNameToken = varNameToken;
        }

        public override string ToString()
        {
            return string.Format("({0})", varNameToken);
        }
    }

    public class BinaryOperationNode : NodeBase
    {
        public NodeBase leftNode { get; protected set; }
        public Token operatorToken { get; protected set; }
        public NodeBase rightNode { get; protected set; }

        public BinaryOperationNode(NodeBase leftNode, Token operatorToken, NodeBase rightNode)
            : base(leftNode.startPosition, rightNode.endPosition)
        {
            this.leftNode = leftNode;
            this.operatorToken = operatorToken;
            this.rightNode = rightNode;
        }

        public override string ToString()
        {
            return string.Format("({0} {1} {2})", leftNode, operatorToken, rightNode);
        }
    }

    public class UnaryOperationNode : NodeBase
    {
        public Token operatorToken { get; protected set; }
        public NodeBase operandNode { get; protected set; }

        public UnaryOperationNode(Token operatorToken, NodeBase operandNode)
            : base(operatorToken.startPosition, operandNode.endPosition)
        {
            this.operatorToken = operatorToken;
            this.operandNode = operandNode;
        }
    }

    public class IfNode : NodeBase
    {
        public (NodeBase, NodeBase)[] caseNodes { get; protected set; }
        public NodeBase elseNode { get; protected set; }

        public IfNode((NodeBase, NodeBase)[] caseNodes)
            : base(caseNodes[0].Item1.startPosition, caseNodes[^1].Item2.endPosition)
        {
            this.caseNodes = caseNodes;
        }
        public IfNode((NodeBase, NodeBase)[] caseNodes, NodeBase elseNode)
            : base(caseNodes[0].Item1.startPosition, elseNode.endPosition)
        {
            this.caseNodes = caseNodes;
            this.elseNode = elseNode;
        }
    }

    public class ForNode : NodeBase
    {
        public Token varNameToken { get; protected set; }
        public NodeBase startValueNode { get; protected set; }
        public NodeBase endValueNode { get; protected set; }
        public NodeBase stepValueNode { get; protected set; }
        public NodeBase bodyNode { get; protected set; }
        public bool shouldReturnNull { get; protected set; }

        public ForNode(Token varNameToken,
                       NodeBase startValueNode,
                           NodeBase endValueNode,
                           NodeBase stepValueNode,
                           NodeBase bodyNode,
                           bool shouldReturnNull)
            : base(varNameToken.startPosition, bodyNode.endPosition)
        {
            this.varNameToken = varNameToken;
            this.startValueNode = startValueNode;
            this.endValueNode = endValueNode;
            this.stepValueNode = stepValueNode;
            this.bodyNode = bodyNode;
            this.shouldReturnNull = shouldReturnNull;
        }
    }

    public class WhileNode : NodeBase
    {
        public NodeBase conditionNode { get; protected set; }
        public NodeBase bodyNode { get; protected set; }
        public bool shouldReturnNull { get; protected set; }

        public WhileNode(NodeBase conditionNode, NodeBase bodyNode, bool
            ↪ shouldReturnNull)
            : base(conditionNode.startPosition, bodyNode.endPosition)
        {
            this.conditionNode = conditionNode;
            this.bodyNode = bodyNode;
            this.shouldReturnNull = shouldReturnNull;
        }
    }

    public class FunctionDefinitionNode : NodeBase
    {
        public Token varNameToken { get; protected set; }
        public Token[] argNameTokens { get; protected set; }
        public NodeBase bodyNode { get; protected set; }
        public bool shouldAutoReturn { get; protected set; }

        public FunctionDefinitionNode(Token varNameToken,
                                      Token[] argNameTokens,
                                      NodeBase bodyNode,
                                      bool shouldAutoReturn)
            : base(varNameToken.startPosition, bodyNode.endPosition)
        {
            this.varNameToken = varNameToken;
            this.argNameTokens = argNameTokens;
            this.bodyNode = bodyNode;
            this.shouldAutoReturn = shouldAutoReturn;
        }
    }

    public class CallNode : NodeBase
    {
        public NodeBase nodeToCall { get; protected set; }
        public NodeBase[] argumentNodes { get; protected set; }

        public CallNode(NodeBase nodeToCall, NodeBase[] argumentNodes)
            : base(nodeToCall.startPosition, (argumentNodes.Length > 0) ?
                ↪ argumentNodes[^1].endPosition : nodeToCall.endPosition)
        {
            this.nodeToCall = nodeToCall;
            this.argumentNodes = argumentNodes;
            }
        }

        public class ReturnNode : NodeBase
        {
            public NodeBase nodeToReturn { get; protected set; }

            public ReturnNode(Position startPosition, Position endPosition)
                : base(startPosition, endPosition) { }
            public ReturnNode(NodeBase nodeToReturn)
                : base(nodeToReturn.startPosition, nodeToReturn.endPosition)
            {
                this.nodeToReturn = nodeToReturn;
            }
        }

        public class ContinueNode : NodeBase
        {
            public ContinueNode(Position startPosition, Position endPosition)
                : base(startPosition, endPosition) { }
        }

        public class BreakNode : NodeBase
        {
            public BreakNode(Position startPosition, Position endPosition)
                : base(startPosition, endPosition) { }
        }
}
```

Parser.cs

```
namespace ShorkSharp
{
    public class Parser
    {
        Token[] tokens;
        int tokenIndex = 0;
        Token currentToken;

        public Parser(Token[] tokens)
        {
            this.tokens = tokens;
            this.currentToken = this.tokens[0];
        }

        Token Advance()
        {
            tokenIndex++;
            currentToken = (tokenIndex < tokens.Length) ?
                ↪ this.tokens[tokenIndex] : null;
            return currentToken;
        }

        Token Reverse(int amount = 1)
        {
            tokenIndex -= amount;
            currentToken = (tokenIndex < tokens.Length) ?
                ↪ this.tokens[tokenIndex] : null;
            return currentToken;
        }

        public ParseResult Parse()
        {
            ParseResult result = ParseStatements();

            if (result.error != null && currentToken.type != TokenType.EOF)
                return result.Failure(new InvalidSyntaxError("Unexpected␣EOF",
                    ↪ currentToken.startPosition));

            return result;
        }

        //##################################

        protected ParseResult ParseStatements()
        {
            ParseResult result = new ParseResult();
            List<NodeBase> statements = new List<NodeBase>();
            Position startPosition = currentToken.startPosition.Copy();

            while (currentToken.type != TokenType.NEWLINE)
            {
                result.RegisterAdvancement();
                Advance();
            }

            NodeBase statement = result.Register(ParseStatement());
            if (result.error != null)
                return result;
            statements.Add(statement);

            bool hasMoreStatements = true;
            while (true)
            {
                int newlineCount = 0;
                while (currentToken.type == TokenType.NEWLINE)
                {
                    result.RegisterAdvancement();
                    Advance();
                    newlineCount++;
                }
                if (newlineCount == 0)
                    hasMoreStatements = false;

                if (!hasMoreStatements)
                    break;

                statement = result.TryRegister(ParseStatement());
                if (statement == null)
                {
                    Reverse(result.toReverseCount);
                    hasMoreStatements = false;
                    continue;
                }
                statements.Add(statement);
            }

            return result.Success(new CodeBlockNode(statements, startPosition,
                ↪ currentToken.endPosition));
        }

        protected ParseResult ParseStatement()
        {
            ParseResult result = new ParseResult();
            Position startPosition = currentToken.startPosition.Copy();

        if (currentToken.Matches(TokenType.KEYWORD, "return"))
        {
            result.RegisterAdvancement();
            Advance();

            NodeBase expression = result.TryRegister(ParseExpression());
            if (expression == null)
            {
                Reverse(result.toReverseCount);
                return result.Success(new ReturnNode(startPosition, currentToken.endPosition));
            }
            else
                return result.Success(new ReturnNode(expression));
        }

        else if (currentToken.Matches(TokenType.KEYWORD, "continue"))
        {
            result.RegisterAdvancement();
            Advance();
            return result.Success(new ContinueNode(startPosition, currentToken.endPosition));
        }

        else if (currentToken.Matches(TokenType.KEYWORD, "break"))
        {
            result.RegisterAdvancement();
            Advance();
            return result.Success(new BreakNode(startPosition, currentToken.endPosition));
        }

        else
        {
            NodeBase expression = result.Register(ParseExpression());
            if (result.error != null)
                return result.Failure(new InvalidSyntaxError("Expected␣'RETURN',␣'CONTINUE',␣'BREAK',␣'VAR',␣'IF',␣'FOR',␣'WHILE',␣'FUN',␣int,␣float,␣identifier,␣'+',␣'-',␣'(',␣'['␣or␣'NOT'", currentToken.startPosition));

            return result.Success(expression);
        }
    }

    protected ParseResult ParseExpression()
    {
        ParseResult result = new ParseResult();

        if (currentToken.Matches(TokenType.KEYWORD, "var"))
        {
            result.RegisterAdvancement();
            Advance();

            if (currentToken.type != TokenType.IDENTIFIER)
                return result.Failure(new InvalidSyntaxError("Expected␣identifier", currentToken.startPosition));

            Token varNameToken = currentToken;
            result.RegisterAdvancement();
            Advance();
```

```csharp

                if (currentToken.type != TokenType.EQUALS)
                    return result.Failure(new InvalidSyntaxError("Expected '='", currentToken.startPosition));

                result.RegisterAdvancement();
                Advance();

                NodeBase expression = result.Register(ParseExpression());
                if (result.error != null) return result;
                return result.Success(new VarAssignNode(varNameToken, expression));
            }

            else
            {
                NodeBase node = result.Register(ParseBinaryOperation(ParseComparisonExpression, new (TokenType, string)[] { (TokenType.KEYWORD, "and"), (TokenType.KEYWORD, "or") }));
                if (result.error != null)
                    return result.Failure(new InvalidSyntaxError("Expected 'VAR', 'IF', 'FOR', 'WHILE', 'FUNC', number, identifier, '+', '-', '(', '[' or 'NOT'", currentToken.startPosition));
                return result.Success(node);
            }
        }

        protected ParseResult ParseComparisonExpression()
        {
            ParseResult result = new ParseResult();
            NodeBase node;

            if (currentToken.Matches(TokenType.KEYWORD, "not"))
            {
                Token operatorToken = currentToken;
                result.RegisterAdvancement();
                Advance();

                node = result.Register(ParseComparisonExpression());
                if (result.error != null) return result;
                return result.Success(node);
            }

            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 }));
            if (result.error != null)
                return result.Failure(new InvalidSyntaxError("Expected number, identifier, '+', '-', '(', '[', 'IF', 'FOR', 'WHILE', 'FUNC' or 'NOT'", currentToken.startPosition));
            return result.Success(node);
        }

        protected ParseResult ParseArithmaticExpression()
        {
            return ParseBinaryOperation(ParseTerm, new TokenType[] { TokenType.PLUS, TokenType.MINUS });
        }

        protected ParseResult ParseTerm()
        {
            return ParseBinaryOperation(ParseFactor, new TokenType[] {
                ↪ TokenType.MULTIPLY, TokenType.DIVIDE });
        }

        protected ParseResult ParseFactor()
        {
            ParseResult result = new ParseResult();

            if (currentToken.Matches(TokenType.PLUS, TokenType.MINUS))
            {
                Token operandToken = currentToken;
                result.RegisterAdvancement();
                Advance();
                NodeBase factor = result.Register(ParseFactor());
                if (result.error != null) return result;
                return result.Success(new UnaryOperationNode(operandToken,
                    ↪ factor));
            }

            return ParseExponent();
        }

        protected ParseResult ParseExponent()
        {
            return ParseBinaryOperation(ParseCall, new TokenType[] {
                ↪ TokenType.EXPONENT }, ParseFactor);
        }

        protected ParseResult ParseCall()
        {
            ParseResult result = new ParseResult();

             NodeBase atom = result.Register(ParseAtom());
            if (result.error != null) return result;

            if (currentToken.type == TokenType.LPAREN)
            {
                result.RegisterAdvancement();
                Advance();

                List<NodeBase> args = new List<NodeBase>();

                if (currentToken.type == TokenType.RPAREN)
                {
                    result.RegisterAdvancement();
                    Advance();
                }
                else
                {
                    args.Add(result.Register(ParseExpression()));
                    if (result.error != null)
                        return result.Failure(new InvalidSyntaxError("Expected␣
                        ↪ ')',␣'VAR',␣'IF',␣'FOR',␣'WHILE',␣'FUNC',␣number,␣
                        ↪ identifier,␣'+',␣'-',␣'(',␣'['␣or␣'NOT'",
                        ↪ currentToken.startPosition));

                    while (currentToken.type == TokenType.COMMA)
                    {
                        result.RegisterAdvancement();
                        Advance();

                        args.Add(result.Register(ParseExpression()));
                        if (result.error != null) return result;
                    }

                    if (currentToken.type != TokenType.RPAREN)
                        return result.Failure(new InvalidSyntaxError("Expected ',' or ')'", currentToken.startPosition));

                    result.RegisterAdvancement();
                    Advance();
                }

                return result.Success(new CallNode(atom, args.ToArray()));
            }
            return result.Success(atom);
        }

        protected ParseResult ParseAtom()
        {
            ParseResult result = new ParseResult();

            if (currentToken.type == TokenType.NUMBER)
            {
                result.RegisterAdvancement();
                Advance();
                return result.Success(new NumberNode(currentToken));
            }

            else if (currentToken.type == TokenType.STRING)
            {
                result.RegisterAdvancement();
                Advance();
                return result.Success(new StringNode(currentToken));
            }

            else if (currentToken.type == TokenType.IDENTIFIER)
            {
                result.RegisterAdvancement();
                Advance();
                return result.Success(new VarAccessNode(currentToken));
            }

            else if (currentToken.type == TokenType.LPAREN)
            {
                result.RegisterAdvancement();
                Advance();

                NodeBase expression = result.Register(ParseExpression());
                if (result.error != null) return result;

                if (currentToken.type == TokenType.RPAREN)
                {
                    result.RegisterAdvancement();
                    Advance();
                    return result.Success(expression);
                }
                else return result.Failure(new InvalidSyntaxError("Expected ')'", currentToken.startPosition));
            }

            else if (currentToken.type == TokenType.LBRACKET)
            {
                NodeBase list = result.Register(ParseListExpression());
                if (result.error != null) return result;
                return result.Success(list);
            }

            else if (currentToken.Matches(TokenType.KEYWORD, "if"))
            {
                NodeBase ifNode = result.Register(ParseIfExpression());
                if (result.error != null) return result;
                return result.Success(ifNode);
            }

            else if (currentToken.Matches(TokenType.KEYWORD, "for"))
            {
                NodeBase forNode = result.Register(ParseForExpression());
                if (result.error != null) return result;
                return result.Success(forNode);
            }

            else if (currentToken.Matches(TokenType.KEYWORD, "while"))
            {
                NodeBase whileNode = result.Register(ParseWhileExpression());
                if (result.error != null) return result;
                return result.Success(whileNode);
            }

            else if (currentToken.Matches(TokenType.KEYWORD, "func"))
            {
                NodeBase functionDefinition =
                    ↪ result.Register(ParseFunctionDefinition());
                if (result.error != null) return result;
                return result.Success(functionDefinition);
            }

            else return result.Failure(new InvalidSyntaxError("Expected␣number,␣
                ↪ identifier,␣'+',␣'-',␣'(',␣'[',␣IF',␣'FOR',␣'WHILE',␣'FUNC'",
                ↪ currentToken.startPosition));
        }

        protected ParseResult ParseListExpression()
        {
            ParseResult result = new ParseResult();

            List<NodeBase> elements = new List<NodeBase>();
            Position startPosition = currentToken.startPosition.Copy();

            if (currentToken.type != TokenType.LBRACKET)
                return result.Failure(new InvalidSyntaxError("Expected␣'['",
                    ↪ currentToken.startPosition));

            result.RegisterAdvancement();
            Advance();

            if (currentToken.type == TokenType.RBRACKET)
            {
                result.RegisterAdvancement();
                Advance();
            }
            else
```

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

                while (currentToken.type == TokenType.COMMA)
                {
                    result.RegisterAdvancement();
                    Advance();

                    elements.Add(result.Register(ParseExpression()));
                    if (result.error != null) return result;
                }

                if (currentToken.type != TokenType.RBRACKET)
                    return result.Failure(new InvalidSyntaxError("Expected␣']'",
                        ↪ currentToken.startPosition));

                result.RegisterAdvancement();
                Advance();
            }

            return result.Success(new ListNode(elements, startPosition,
                ↪ currentToken.endPosition));
        }

        // TODO: ParseIfExpression
        protected ParseResult ParseIfExpression()
        {
            throw new NotImplementedException();
        }

        protected ParseResult ParseForExpression()
        {
            ParseResult result = new ParseResult();

            if (!currentToken.Matches(TokenType.KEYWORD, "for"))
                return result.Failure(new InvalidSyntaxError("Expected␣'FOR'",
                    ↪ currentToken.startPosition));

            result.RegisterAdvancement();
            Advance();

            if (currentToken.type != TokenType.IDENTIFIER)
                return result.Failure(new InvalidSyntaxError("Expected␣
                    ↪ identifier", currentToken.startPosition));

            Token varNameToken = currentToken;
            result.RegisterAdvancement();
            Advance();

            if (currentToken.type != TokenType.EQUALS)
                return result.Failure(new InvalidSyntaxError("Expected␣'='",
                    ↪ currentToken.startPosition));

            result.RegisterAdvancement();
            Advance();

            NodeBase startValue = result.Register(ParseExpression());
            if (result.error != null) return result;

            if (!currentToken.Matches(TokenType.KEYWORD, "to"))
                return result.Failure(new InvalidSyntaxError("Expected␣'TO'",
                    ↪ currentToken.startPosition));

            result.RegisterAdvancement();
            Advance();

            NodeBase endValue = result.Register(ParseExpression());
            if (result.error != null) return result;

            NodeBase stepValue = null;
            if (currentToken.Matches(TokenType.KEYWORD, "step"))
            {
                result.RegisterAdvancement();
                Advance();

                stepValue = result.Register(ParseExpression());
                if (result.error != null) return result;
            }

            if (!currentToken.Matches(TokenType.KEYWORD, "do"))
                return result.Failure(new InvalidSyntaxError("Expected␣'DO'",
                    ↪ currentToken.startPosition));

            result.RegisterAdvancement();
            Advance();

            if (currentToken.type == TokenType.NEWLINE)
            {
                result.RegisterAdvancement();
                Advance();

                NodeBase bodyNodes = result.Register(ParseStatements());
                if (result.error != null) return result;

                if (!currentToken.Matches(TokenType.KEYWORD, "end"))
                    return result.Failure(new InvalidSyntaxError("Expected␣
                        ↪ 'END'", currentToken.startPosition));

                result.RegisterAdvancement();
                Advance();

                return result.Success(new ForNode(varNameToken, startValue,
                    ↪ endValue, stepValue, bodyNodes, true));
            }

            NodeBase bodyNode = result.Register(ParseStatement());
            if (result.error != null) return result;

            return result.Success(new ForNode(varNameToken, startValue,
                ↪ endValue, stepValue, bodyNode, false));
        }

        protected ParseResult ParseWhileExpression()
        {
            ParseResult result = new ParseResult();

            if (!currentToken.Matches(TokenType.KEYWORD, "while"))
                return result.Failure(new InvalidSyntaxError("Expected␣'WHILE'",
                    ↪ currentToken.startPosition));

            result.RegisterAdvancement();
            Advance();

            NodeBase condition = result.Register(ParseExpression());
            if (result.error != null) return result;

            if (!currentToken.Matches(TokenType.KEYWORD, "do"))
                return result.Failure(new InvalidSyntaxError("Expected 'do'", currentToken.startPosition));

            result.RegisterAdvancement();
            Advance();

            NodeBase bodyNode;
            if (currentToken.type == TokenType.NEWLINE)
            {
                result.RegisterAdvancement();
                Advance();

                bodyNode = result.Register(ParseStatements());
                if (result.error != null) return result;

                if (!currentToken.Matches(TokenType.KEYWORD, "end"))
                    return result.Failure(new InvalidSyntaxError("Expected 'END'", currentToken.startPosition));

                result.RegisterAdvancement();
                Advance();

                return result.Success(new WhileNode(condition, bodyNode, true));
            }
            else
            {
                bodyNode = result.Register(ParseStatement());
                if (result.error != null) return result;

                return result.Success(new WhileNode(condition, bodyNode, false));
            }
        }

        protected ParseResult ParseFunctionDefinition()
        {
            throw new NotImplementedException();
        }

        //########################################

        protected delegate ParseResult BinaryOperationDelegate();
        protected ParseResult ParseBinaryOperation(BinaryOperationDelegate leftFunc, TokenType[] operations)
        {
            return ParseBinaryOperation(leftFunc, operations, leftFunc);
        }
        protected ParseResult ParseBinaryOperation(BinaryOperationDelegate leftFunc, TokenType[] operations, BinaryOperationDelegate rightFunc)
        {
            ParseResult result = new ParseResult();

            NodeBase leftNode = result.Register(leftFunc());
            if (result.error != null)
```

```
                return result;

            while (operations.Contains(currentToken.type))
            {
                Token operatorToken = currentToken;
                result.RegisterAdvancement();
                Advance();

                NodeBase rightNode = result.Register(rightFunc());
                if (result.error != null)
                    return result;

                leftNode = new BinaryOperationNode(leftNode, operatorToken, rightNode);
            }

            return result.Success(leftNode);
        }
        protected ParseResult ParseBinaryOperation(BinaryOperationDelegate leftFunc, (TokenType, string)[] operations)
        {
            return ParseBinaryOperation(leftFunc, operations, leftFunc);
        }
        protected ParseResult ParseBinaryOperation(BinaryOperationDelegate leftFunc, (TokenType, string)[] operations, BinaryOperationDelegate rightFunc)
        {
            ParseResult result = new ParseResult();

            NodeBase leftNode = result.Register(leftFunc());
            if (result.error != null)
                return result;

            while (operations.Contains((currentToken.type, (string)currentToken.value)))
            {
                Token operatorToken = currentToken;
                result.RegisterAdvancement();
                Advance();

                NodeBase rightNode = result.Register(rightFunc());
                if (result.error != null)
                    return result;

                leftNode = new BinaryOperationNode(leftNode, operatorToken, rightNode);
            }

            return result.Success(leftNode);
        }
    }
}
```

ParseResult.cs

```
namespace ShorkSharp
{
    public class ParseResult
    {
        public ShorkError error { get; protected set; }
        public NodeBase node { get; protected set; }
        public int advanceCount { get; protected set; } = 0;
        public int lastAdvanceCount { get; protected set; } = 0;
        public int toReverseCount { get; protected set; } = 0;

        public ParseResult() { }

        public void RegisterAdvancement()
        {
            lastAdvanceCount = 1;
            advanceCount++;
        }

        public NodeBase Register(ParseResult result)
        {
            lastAdvanceCount = result.advanceCount;
            this.advanceCount += result.advanceCount;
            if (result.error != null) this.error = result.error;
            return result.node;
        }

        public NodeBase TryRegister(ParseResult result)
        {
            if (result.error != null)
            {
                toReverseCount = result.advanceCount;
                return null;
            }
            return Register(result);
        }

        public ParseResult Success(NodeBase node)
        {
            this.node = node;
            return this;
        }

        public ParseResult Failure(ShorkError error)
        {
            if (this.error == null || this.lastAdvanceCount == 0)
                this.error = error;
            return this;
        }
    }
}
```

Lexer.cs

```
namespace ShorkSharp
{
    /// <summary>
    /// The lexer takes in the input text and converts it into a series of
        ↪ tokens.
    /// </summary>
    public class Lexer
    {
        /// <summary>
        /// The words recognised as keywords.
        /// </summary>
        static readonly string[] KEYWORDS =
        {
            "var",
            "and",
            "or",
            "not",
            "if",
            "then",
            "elif",
            "else",
            "for",
            "to",
            "step",
            "func",
            "while",
            "do",
            "end",
            "return",
            "continue",
            "break"
        };
        static readonly char[] WHITESPACE = { ' ', '\t', '\r' };
        static readonly char[] DIGITS = { '0', '1', '2', '3', '4', '5', '6',
            ↪ '7', '8', '9' };
        static readonly char[] DIGITS_WITH_DOT = DIGITS.Concat(new char[] { '.'
            ↪ }).ToArray();
        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' };
        static readonly char[] LETTERS_WITH_UNDERSCORE = LETTERS.Concat(new
            ↪ char[] { '_' }).ToArray();

        public Position position { get; protected set; }
        public string input { get; protected set; }
        public char currentChar { get; protected set; } = '\0';

        public Lexer(string input)
        {
            this.input = input;
            this.position = new Position(input);
        }
        public Lexer(string input, string filename)
        {
            this.input = input;
            this.position = new Position(filename);
        }

        void Advance()
        {
            position.Advance(currentChar);

            if (position.index < input.Length)
                currentChar = input[position.index];
            else
                currentChar = '\0';
        }

        /// <summary>
        /// Runs the lexer and returns the result.
        /// </summary>
        /// <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>
        public (Token[], ShorkError?) Lex()
        {
            if (input.Length == 0)
                return (new Token[] { }, new ShorkError("Empty␣Input", "Input␣
                   ↪ text␣is␣empty", null));
            this.currentChar = input[0];

            List<Token> tokens = new List<Token>();

            while (currentChar != '\0')
            {
                if (WHITESPACE.Contains(currentChar))
                {
                    Advance();
                }

                // Number Tokens
                else if (DIGITS.Contains(currentChar))
                {
                    tokens.Add(MakeNumberToken());
                }

                // String Tokens
                else if (currentChar == '"')
                {
                    (Token token, ShorkError error) = MakeStringToken();
                    if (error != null)
                        return (null, error);
                    tokens.Add(token);
                }

                // Identifiers and Keywords
                else if (LETTERS.Contains(currentChar))
                {
                    tokens.Add(MakeIdentifierToken());
                }

                // Simple tokens
                else
                {
                    switch (currentChar)
                    {
                        default:
                            return (new Token[] { },
                                    new
                                       ↪ InvalidCharacterError(string.Format("'{0}'",
                                       ↪ currentChar), position));
                        case '+':
                            tokens.Add(new Token(TokenType.PLUS, position));
                            Advance();
                            break;
                        case '-':
                            TokenType ttype = TokenType.MINUS;
                            Position startPosition = position.Copy();
                            Advance();

                            if (currentChar == '>')
                            {
                                ttype = TokenType.ARROW;
                                Advance();
                            }

                            tokens.Add(new Token(ttype, startPosition,
                               ↪ position));
                            break;
                    case '*':
                        tokens.Add(new Token(TokenType.MULTIPLY, position));
                        Advance();
                        break;
                    case '/':
                        tokens.Add(new Token(TokenType.DIVIDE, position));
                        Advance();
                        break;
                    case '^':
                        tokens.Add(new Token(TokenType.EXPONENT, position));
                        Advance();
                        break;

                    case '!':
                        (Token token, ShorkError error) =
                            ↪ MakeNotEqualsToken();
                        if (error != null) return (null, error);
                        tokens.Add(token);
                        break;
                    case '=':
                        tokens.Add(MakeEqualsToken());
                        break;
                    case '<':
                        tokens.Add(MakeLessThanToken());
                        break;
                    case '>':
                        tokens.Add(MakeGreaterThanToken());
                        break;

                    case '.':
                        tokens.Add(new Token(TokenType.DOT, position));
                        Advance();
                        break;
                    case ',':
                        tokens.Add(new Token(TokenType.COMMA, position));
                        Advance();
                        break;

                    case '(':
                        tokens.Add(new Token(TokenType.LPAREN, position));
                        Advance();
                        break;
                    case ')':
                        tokens.Add(new Token(TokenType.RPAREN, position));
                        Advance();
                        break;
                    case '{':
                        tokens.Add(new Token(TokenType.LBRACE, position));
                        Advance();
                        break;
                    case '}':
                        tokens.Add(new Token(TokenType.RBRACE, position));
                        Advance();
                        break;
                    case '[':
                        tokens.Add(new Token(TokenType.LBRACKET, position));
                        Advance();
                        break;
                    case ']':
                        tokens.Add(new Token(TokenType.RBRACKET, position));
                        Advance();
                        break;
                }
            }
        }

        return (tokens.ToArray(), null);
    }

    Token MakeNumberToken()
    {
        string numstring = string.Empty + currentChar;
        bool hasDecimalPoint = false;
        Position startPosition = position.Copy();

        Advance();
        while (DIGITS_WITH_DOT.Contains(currentChar))
        {
            if (currentChar == '.')
            {
                if (hasDecimalPoint)
                    break;
                else
                    hasDecimalPoint = true;
            }
            numstring += currentChar;
            Advance();
        }

        return new Token(TokenType.NUMBER, decimal.Parse(numstring), startPosition, position);
    }

    (Token, ShorkError) MakeStringToken()
    {
        Position startPosition = position.Copy();
        string str = string.Empty;
        Advance();

        bool escaping = false;
        while (true)
        {
            if (escaping)
            {
                switch (currentChar)
                {
                    default:
                        return (null, new InvalidEscapeSequenceError(string.Format("\\{0}", currentChar), position));
                    case '"':
                        str += '"';
                        break;
                    case '\\':
                        str += '\\';
                        break;
                    case 't':
                        str += '\t';
                        break;
                }
                escaping = false;
            }

            else if (currentChar == '"')
            {
                Advance();
                break;
            }

            else if (currentChar == '\\')
                escaping = true;

            else
                str += currentChar;

            Advance();
        }

        return (new Token(TokenType.STRING, str, startPosition, position), null);
    }

    Token MakeIdentifierToken()
    {
        Position startPosition = position.Copy();
        string idstr = string.Empty + currentChar;
        Advance();

        while (LETTERS_WITH_UNDERSCORE.Contains(currentChar))
        {
            idstr += currentChar;
            Advance();
        }

        if (idstr == "true")
            return new Token(TokenType.BOOL, true, startPosition, position);
        else if (idstr == "false")
            return new Token(TokenType.BOOL, false, startPosition, position);
        else if (idstr == "null")
            return new Token(TokenType.NULL, startPosition, position);
        else
        {
            TokenType ttype = KEYWORDS.Contains(idstr.ToLower()) ? TokenType.KEYWORD : TokenType.IDENTIFIER;
            return new Token(ttype, idstr, startPosition, position);
        }
    }

    Token MakeEqualsToken()
    {
        Position startPosition = position.Copy();
        TokenType ttype = TokenType.EQUALS;
        Advance();
        if (currentChar == '=')
        {
            ttype = TokenType.DOUBLE_EQUALS;
            Advance();
        }
        return new Token(ttype, startPosition, position);
    }

    (Token, ShorkError) MakeNotEqualsToken()
    {
        Position startPosition = position.Copy();
        Advance();
        if (currentChar == '=')
        {
            Advance();
            return (new Token(TokenType.NOT_EQUALS, startPosition, position), null);
        }
        return (null, new InvalidCharacterError("", position));
    }

    Token MakeLessThanToken()
    {
        Position startPosition = position.Copy();
        TokenType ttype = TokenType.LESS_THAN;
        Advance();
        if (currentChar == '=')
        {
            ttype = TokenType.LESS_THAN_OR_EQUAL;
            Advance();
        }
        return new Token(ttype, startPosition, position);
    }

    Token MakeGreaterThanToken()
    {
        Position startPosition = position.Copy();
        TokenType ttype = TokenType.GREATER_THAN;
        Advance();
        if (currentChar == '=')
        {
            ttype = TokenType.GREATER_THAN_OR_EQUAL;
            Advance();
        }
        return new Token(ttype, startPosition, position);
    }
}
}
```

ShorkError.cs

```
namespace ShorkSharp
{
    public class ShorkError
    {
        public string errorName { get; protected set; }
        public string details { get; protected set; }

        public Position startPosition { get; protected set; }

        public ShorkError(string errorName, string details, Position startPosition)
        {
            this.errorName = errorName;
            this.details = details;
            this.startPosition = startPosition;
        }

        public override string ToString()
        {
            string output = string.Format("{0}: {1}", errorName, details);

            if (startPosition != null)
                output += string.Format("\nFile: '{0}', line {1}", startPosition.filename, startPosition.line+1);

            return output;
        }
    }

    public class InvalidCharacterError : ShorkError
    {
        public InvalidCharacterError(string details, Position startPosition)
            : base("Invalid␣Character", details, startPosition) { }
    }

    public class InvalidSyntaxError : ShorkError
    {
        public InvalidSyntaxError(string details, Position startPosition)
            : base("Invalid␣Syntax", details, startPosition) { }
    }

    public class InvalidEscapeSequenceError : ShorkError
    {
        public InvalidEscapeSequenceError(string details, Position startPosition)
            : base("Invalid␣Escape␣Sequence", details, startPosition) { }
    }
}
```

Token.cs

```
namespace ShorkSharp
{
    public class Token
    {
        public TokenType type { get; protected set; }
        public dynamic value { get; protected set; }

        public Position startPosition { get; protected set; }
        public Position endPosition { get; protected set; }

        public Token(TokenType type, Position startPosition)
        {
            this.type = type;
            this.value = null;
            this.startPosition = startPosition.Copy();
            this.endPosition = startPosition.Copy();
        }
        public Token(TokenType type, Position startPosition, Position
            ↪ endPosition)
        {
            this.type = type;
            this.value = null;
            this.startPosition = startPosition.Copy();
            this.endPosition = endPosition.Copy();
        }
        public Token(TokenType type, dynamic value, Position startPosition)
        {
            this.type = type;
            this.value = value;
            this.startPosition = startPosition.Copy();
            this.endPosition = startPosition.Copy();
        }
        public Token(TokenType type, dynamic value, Position startPosition,
            ↪ Position endPosition)
        {
            this.type = type;
            this.value = value;
            this.startPosition = startPosition.Copy();
            this.endPosition = endPosition.Copy();
        }

        public bool Matches(params TokenType[] types)
        {
            foreach (TokenType ttpe in types)
                if (this.type == type) return true;
            return this.type == type;
        }
        public bool Matches(TokenType type, dynamic value)
        {
            if (type == TokenType.KEYWORD)
                return this.type == type && ((string)this.value).ToLower() ==
                    ↪ ((string)value).ToLower();
            return this.type == type && this.value == value;
        }

        public override string ToString()
        {
            if (value == null)
                return string.Format("[{0}]", type);
            else
                return string.Format("[{0}␣:␣{1}]", type, value);
        }
    }
}
```

TokenType.cs

```
namespace ShorkSharp
{
    public enum TokenType
    {
        NUMBER,
        STRING,
        BOOL,
        NULL,

        KEYWORD,
        IDENTIFIER,

        PLUS,
        MINUS,
        MULTIPLY,
        DIVIDE,
        EXPONENT,

        EQUALS,
        DOUBLE_EQUALS,
        NOT_EQUALS,
        LESS_THAN,
        GREATER_THAN,
        LESS_THAN_OR_EQUAL,
        GREATER_THAN_OR_EQUAL,

        DOT,
        COMMA,
        ARROW,

        LPAREN,
        RPAREN,
        LBRACE,
        RBRACE,
        LBRACKET,
        RBRACKET,

        NEWLINE,
        EOF
    }
}
```