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

```
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 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("(List{{{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;
        }
    }
}
```

Listing 2: 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];
        }

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

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

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

            return result;
        }

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

        protected ParseResult ParseExpression()
        {
            throw new NotImplementedException();
        }
    }
}
```

Listing 3: ParseResult.cs

```
namespace ShorkSharp
{
    public class ParseResult
    {
        public ShorkError error { get; protected set; }
        public NodeBase node { get; protected set; }
```

```csharp
        public int advanceCount { get; protected set; } = 0;

        public ParseResult() { }

        public void RegisterAdvancement()
        {
            advanceCount++;
        }

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

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

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

Listing 4: Lexer.cs

```csharp
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);
    }
}
}
```

Listing 5: 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) { }
    }
}
```

Listing 6: 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 override string ToString()
        {
            if (value == null)
                return string.Format("[{0}]", type);
            else
                return string.Format("[{0}␣:␣{1}]", type, value);
        }
    }
}
```

Listing 7: 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
    }
}
```