Reduced Triangle Compiler Report

Jack Neilson

November 5, 2017

1 Introduction

The goal of this program is to take a source file written in a reduced form of the triangle langauge and compile it in to a source language. This first submission deals with only the scanner and parser parts of the semantic analyser. The program has been implemented in C# using .NET Core.

2 Implementation

2.1 Scanner

The scanner reads the source file a single character at a time and groups them in to tokens, ignoring white space and comments, with a single class "Scanner.cs". It returns an enumerable containing all tokens to be iterated over by the parser. A regular expression matching any character in the alphabet is used to generate identifier tokens (see example 1). Any other single-character special tokens (numerals, operators, EOF) have their own, seperate logic in a switch statement to generate the appropriate token (see example 2).

2.2 Parser

The parser takes an enumerable of all the tokens that have been generated by the scanner and will iterate through them, generating an abstract syntax tree. For now, it simply iterates through them to check for syntax errors. The parser class is split in to several partial classes corresponding to every non-terminal symbol in the language definition. In each of these classes a seperate method is defined for every production rule (see example 3). To solve the problem of the trailing else statement / empty single command, several cases have been defined to simply break in the parseSingleCommand method (see example 4).

Error reporting is done via an ErrorReporter class, which is passed to the Parser when the Parser class is instantiated. The only time the ReportError method needs to be called is in the Accept method, as this is the method that is called when the parser has reached the bottom node of the tree (see example 5), and in partial classes where no descent is needed (e.g. the Declaration partial class, see example 6).

Some classes may call themselves recursively to maintain the tree structure, for example the Commands.cs class contains the ParseSingleCommand method which will call ParseCommand (and therefore ParseSignleCommand) if the current token is "Begin" (see example 7).

The production rules for Expression have been broken up to prevent conflict. The "let" and "if" productions now have a higher priority over the recursive expression production, so there is no conflict over which production to take when receiving this token (see example 8).

Empty parameter sequences may be encountered when calling a method with no arguments. This is dealt with by taking the left parenthesis and then checking if the next token is a right parenthesis. If it is the method simply returns, otherwise the actual parameters are parsed (see example 9).

3 Testing

Testing was done using the triangle files provided, along with the expected output. A small test script has been included which compares the output of scanning and parsing ”test-mini.tri” with the expected output in ”test-mini-output.txt” (conversion from windows to linux line endings and vice-versa may be necessary). If the two outputs are identical, no output should be generated. In the cases where an error is expected to be thrown (test-mini-error.tri and test-mini-error2.tri), output checking was done by hand.

4 Examples

4.1 Example 1

```
Regex token = new Regex("^[a-zA-Z]+");

if (token.IsMatch(((char) _source.Current).ToString())) {
        while (token.IsMatch(((char) _source.Current).ToString())) {
                TakeIt();
    }
        return TokenKind.Identifier;
}
```

4.2 Example 2

```
case '+':
        TakeIt();
        return TokenKind.Operator;
```

4.3 Example 3

```
void ParseCommand() {
        System.Console.WriteLine("Parsing command");
        ParseSingleCommand();
        while (_currentToken.Kind == TokenKind.Semicolon) {
                AcceptIt();
                ParseSingleCommand();
        }
}
```

4.4 Example 4

```
case TokenKind.End:
case TokenKind.Else:
case TokenKind.EndOfText: {
        break;
}
```

4.5 Example 5

```
void Accept(TokenKind expectedKind) {
if (_currentToken.Kind == expectedKind) {
        Token token = _currentToken;
        //_previousLocation = token.Start;
        _tokens.MoveNext();
        _currentToken = _tokens.Current;
} else {
        //In the case of an error, give small message and current token
        //to error reporter
```

```
    _reporter.ReportError("Error line: " + _currentToken.getLine()
                                         + " index: " + _currentToken.getIndex()
                                         + " expected " + expectedKind
                                         + ", got ",
                  _currentToken);
        }
}
```

4.6 Example 6

In this example the only valid tokens are Const and Var, anything else should throw an error.

```
void ParseSingleDeclaration() {
            System.Console.WriteLine("Parsing single declaration");
            switch (_currentToken.Kind) {
                case TokenKind.Const: {
                        ...
                }

                case TokenKind.Var: {
                        ...
                }

                default: {
                    _reporter.ReportError("Error while parsing single" +
                                          "declaration line: "
                                          + _currentToken.getLine()
                                          + " index: "
                                          + _currentToken.getIndex()
                                          + " expected var or const, got ",
                        _currentToken);
                    break;
                }
            }
                }
```

4.7 Example 7

```
void ParseSingleCommand()
        {
        ...
        case TokenKind.Begin:
                    {
                        AcceptIt();
                        ParseCommand();
                        Accept(TokenKind.End);
                        break;
                    }
                }
```

4.8 Example 8

```
void ParseExpression() {
            System.Console.WriteLine("Parsing expression");
            switch (_currentToken.Kind) {
                case TokenKind.Let: {
                                        ...
                }

                case TokenKind.If: {
                                        ...
                }

                default: {
                    ParseSecondaryExpression();
                    break;
                }
            }
        }
```

4.9 Example 9

```
void ParseActualParameterSequence() {
            System.Console.WriteLine("Parsing actual parameter sequence");
            Accept(TokenKind.LeftParen);
            if (_currentToken.Kind == TokenKind.RightParen) {
                AcceptIt();
                return;
            }
            ParseActualParameter();
            while (_currentToken.Kind == TokenKind.Comma) {
                AcceptIt();
                ParseActualParameter();
            }
            Accept(TokenKind.RightParen);
        }
```