Implementation report

Anthony Sébert [1] , 28 october 2018

Disclaimer

Due to lack of time, this report is a bit unconventional. In fact, the information about the coursework is divided in two main parts : the documentation (API), in the folder docs, and the Appendixes (the links in the Scanner and Parser sections of this documents lead to sections of the Appendix). Knowing that I would not have the time to redact a proper report, I have made my code as expressive as possible, and annotated it any time it was necessary. I hope the reader will find it comfortable to read and easier to navigate between the documentation.

Documentation

The documentation has been generated with Doxygen from the in-code documentation. It has been generated in several formats, as listed : HTML, LaTeX, man-pages, rtf and xml. The "main page" of the HTML version is located at docs/html/annotated.html (the real main page is empty, for I had nothing interesting to put there). It also includes beautiful diagrams.

Prerequisites

Microsoft .NET Core 7.0 or later.

Getting started

Decompress the folder, then open a terminal.

```
cd path/to/folder/Compiler
```

Note : if the terminal starts in another drive, just type the name of the drive where the project folder is located, i.e. `D:`

Once you are in the **Compiler** folder, run the program (a sample source code file is already provided).

```
dotnet run source.txt
```

And voilà !

Scanner

Your Scanner (Lexical Analyser) should be developed to read in a source file written in the given source language. The characters in the source file should be compiled into a sequence of recognised language tokens.

During this process your scanner should fulfill the following basic compiler requirements

- [x] Identify and remove whitespace
- [x] Identify and remove language comments
- [x] Identify and produce errors for unknown characters in the language
- [x] Identify and produce errors for unterminated character literals

Parser

- [x] Your Parser (Syntax Analyser) should work on the sequence of tokens created by the scanner. These tokens should be grouped into sentences, creating a set of instructions. The instruction set will represent the purpose of the program defined in the source code
- [x] Your Parser should be able to identify Syntax Errors in the source program and produce errors for instructions that do not conform to the language definition
- [x] On completion of the Parser stage your compiler should maintain an Internal Representation of the instruction set and write out an instruction set, in the correct order, to the console
- [x] Transfer of the data between the `Scanner` and the `Parser`

Appendix

Note : irrelevant parts of the fragments have been omitted so as to lead the eye of the reader to the essential.

Scanner

Fragment 1

```
/**
 * Skips whitespaces and comments in the source file.
 */
protected void IgnoreUseless() {
    while(IsIgnored(source.Current)) {
        switch(source.Current) {
            case ' ':
            case '\t':
            case '\n':
                source.MoveNext();
                break;
```

```
            default:
                source.SkipRestOfLine();
                break;
        }
    }
}
```

Fragment 2

```
/**
 * Determine the token kind to build from the characters processed. Reads the file stream to build the token.
 * @return  a token kind.
 * @see     TokenKind
 */
private TokenKind ScanToken() {
    /* valid characters */
    // ...
    switch(source.Current) {
        /* valid characters */
        // ...
        default:
            TakeIt();
            Compiler.Error(typeof(Scanner).Name, 0, new string[]{
                source._Location.LineNumber.ToString(),
                source._Location.RowNumber.ToString(),
                currentSpelling.ToString()
            }, 1);
            return TokenKind.Error;
    }
}
```

Fragment 3

```
/**
 * Determine the token kind to build from the characters processed. Reads the file stream to build the token.
 * @return  a token kind.
 * @see     TokenKind
 */
private TokenKind ScanToken() {
    /* valid characters */
    // ...
    switch(source.Current) {
        /* other valid characters */
        // ...
        case '\'':
            TakeIt();
            if(source.Current == '\'') {
                TakeIt();
                return TokenKind.CharacterLiteral;
            }
```

```
            else {
                if(IsGraphic(source.Current)) {
                    TakeIt();
                    if(source.Current == '\'') {
                        TakeIt();
                        return TokenKind.CharacterLiteral;
                    }
                }
                TakeIt();
                Compiler.Error(typeof(Scanner).Name, 1, new string[]{
                    source._Location.LineNumber.ToString(),
                    source._Location.RowNumber.ToString(),
                    currentSpelling.ToString()
                }, 1);
                return TokenKind.Error;
            }
        /* unknown characters */
        // ...
    }
}
```

Parser

Fragment 4

```
/**
 * Checks that the given token matches the current stream of tokens, if not prints an error.
 * @param   expectedKinds   an array of expected token kinds.
 */
protected void Accept(TokenKind expectedKind) {
    Location previousLocation = null;
    if(tokens.Current.Kind == expectedKind)
        previousLocation = tokens.Current.Position.Start;
    else
        Compiler.Error(typeof(Parser).Name, 2, new string[]{
            tokens.Current.Position.Start.LineNumber.ToString(),
            tokens.Current.Position.Start.RowNumber.ToString(),
            tokens.Current.Kind.ToString(),
            expectedKind.ToString()
        });
    tokens.MoveNext();
}
```

Fragment 5

```
/**
 * Builds a {@code Compiler} instance. Launches the scanning and the compilation. Prints the tokens representing the source file if the previous operations succeeded.
 * @param   args        command-line one and only argument, the source code file.
 * @see     collection
 */
```

```
public static void Main(string[] args) {
    /* arguments checks */
    // ...

    if(args[0] != null) {
        var compiler = new Compiler(args[0]);
        compiler.collection = compiler.parser.ParseProgram();

        if(0 < compiler.collection.Count) {
            foreach(var element in compiler.collection)
                Info(typeof(Compiler).Name, element.Kind.ToString());
        }
    }
}
```

Fragment 6

```
/**
 * Responsible for the main process of creating a collection of tokens from the
source file.
 * @return  the tokens one at time.
 * @see     Token
 * @see     SourceFile
 */
public IEnumerator<Token> GetEnumerator() {
    Location start = null;
    Token token = null;
    TokenKind kind = 0;
    while(!atEndOfFile) {
        currentSpelling.Clear();
        IgnoreUseless();
        start = source._Location;
        kind = ScanToken();
        token = new Token(kind, currentSpelling.ToString(), new
SourcePosition(start, source._Location));

        if(kind == TokenKind.EndOfText)
            atEndOfFile = true;
        else if(kind == TokenKind.Error)
            Environment.Exit(1);

        if(Debug)
            Compiler.Info(typeof(Scanner).Name, token.ToString());

        yield return token;
    }
}
```

---

1. Computer Science student at The Robert Gordon university (Garthdee House, Garthdee Road, Aberdeen, AB10 7QB, Scotland, United Kingdom)