Release Report

INF01147 - Compilers

Henrique Corrêa Pereira da Silva and Bernardo Hummes Flores

Informatics Institute
Universidade Federal do Rio Grande do Sul

{hcpsilva, bhflores}@inf.ufrgs.br

September 20, 2020

Abstract

In the first stage of the assignment, we've achieved a fully-functioning and correct implementation of a lexical analyzer. Our application utilizes a simple project structure and makes use of several functionalities of the tools we've utilized in order to simplify the token rules and error recognition. The source code also provides simple control over the possible build actions through a full-featured build system specification.

1 Introduction

This report will focus on this course's first objective of the final assignment: a lexical analyzer (or lexer, in short). We shall touch upon the definition itself in Section 2, talk about the tools utilized in Section 3, the development methodology in Section 4 and the overarching conclusion in Section 5.

2 Objective

The first stage outlined a lexical analyzer (lexer) that achieves the following goals to this release:

- Line count;
- Ignore comments;
- Throw lexical errors;
- Match tokens.

The to-be-matched tokens are as follows: reserved keywords, special characters, composite operators, identifiers and literals. While not particularly specific or akin to the ones in the C language, the goal this stage presented would require a minimally complex analyzer.

The tool that would enable us to achieve that is `flex`, the Fast Lexical Analyzer, part of the GNU's toolbox. As presented in class, this tool features a vast array of features that made this task a whole lot easier when using its correct knobs and dials. Yet it is far from the only tool we've used so far in this release, and so we'll show the additional ones in the next Section.

3 Tools

While powerful, `flex` alone doesn't get us too far. As part of the definition, a simple `main` function was provided along with the requirement of a functioning build project. `Make` is by far the simplest to set-up build tool and, therefore, was also defined as the build system to be used.

As to development tools *per-se*, personal preferences spoke louder. As may have been indicated by the source file that generated this document, we've used `Emacs`, but we've also used `Visual Studio Code` as the development environment of choice. Both allow for a fairly unobtrusive shared development footprint because of their focus of not being as fully featured as a simple IDE for the C language. Having that said, we've relied on some `clang` tools of the `LLVM` project, like `clang-format`, to ensure our code was following the same style across all source files.

However, these elements by themselves provide enough breathing room to allow us to define more creatively our project structure. We shall present those choices in the next Section.

4 Methodology

Initially, we defined as essential the ease of use of our project structure of choice and the build system interface to interact with it. That way, we've separated source files, headers, object files and final binaries into different directories, leaving only the essential project-spanning files in the root. Those are the code style rules, the `Makefile` definition and a welcoming `Readme` file.

Using the GNU tool `Make` really helped this organization style, as most other build systems are either too opinionated in that regard or too hard to set-up for such a small project. In its `Makefile` we've defined file-matching rules smart enough to unburden us of the need to specify every header and source-file location manually. That feature in union with a sanitized object-generating logic–separating object files from other based on their module directory–allowed an even faster development cycle, as there was no mental load in that regard. Thus, we've modularized the project having the next iterations in mind and also the respective concern-separation logic of each source file.

After some advancements, we've also decided to create a test-suite, to both ensure safe modifications and also to prepare to the release date. In the matter of what tests to run, the decided inputs were some simple C programs and the `main.c` file of the project itself.

5 Conclusion

The final result of the mentioned development choices led us to a clean and functioning lexer, compliant to the specification and elegant in the use of the available features of the utilized toolset. We believe that those choices will ease our incremental development based on the release schedule of this course and also facilitate remote collaborative development.