Tabulate
Project Report

Anshul Sangrame
CS21BTECH11004

Varun Gupta
CS21BTECH11060

Gautam Singh
CS21BTECH11018

Contents

***Abstract*—This document is a report created by the authors of *Tabulate*, documenting the story behind it, its design and the challenges faced along the way. This report also documents how to build and run the compiler for *Tabulate*.**

1. The Problem Statement

Spreadsheets are an integral part of our lives. Whether it comes to creating timetables, bookkeeping possessions or tabulating marks, it is difficult to imagine life without spreadsheets. Unfortunately, most popular spreadsheet softwares like Microsoft Excel and Google Sheets are *What You See Is What You Get* (WYSIWYG) editors. These softwares do not offer a very good programming interface, which in most cases can automate jobs much faster. Many such solutions are also paid for and not open source entirely. Further, the open source softwares do not have the many features implemented, which makes them unpopular.

That's where Tabulate comes in. With high level programming constructs to abstract the implementation of seemingly complex operations, Tabulate makes it possible to *program* your spreadsheet.

1.1. Our Goals

Tabulate aims to be the go-to Domain Specific Language (DSL) for those who manage many spreadsheets in their everyday life. It aims to automate the tedious process of repetitive entries, updates, and formulae with high performance and efficiency. Built on top of C++, Tabulate aims to provide the programmer with more control over their spreadsheet.

2. Challenges Faced

The authors faced the following challenges in various phases while building *Tabulate*.

2.1. Symbol Table Implementation

The symbol table in *Tabulate* has been implemented using a template class implementation. This enabled the authors to define different symbol tables for symbol table records of various types. However, issues regarding declarations of an identifier in various constructs (for example, declaration of a single identifier as a variable and a struct) came up. Nested declarations also had to be handled, for which a `C++ std::unordered_map` with a `C++ std::stack` was used. Finally, since the symbol table class was a template class, it could not be linked during compilation, and had to be implemented in the same header file itself.

2.2. Input/Output

Tabulate does not aim to be an interactive language. However, it must take input in the form of CSV or similar files for processing. Only CSV file input and output have been implemented by the authors given the time constraints. Implementing other formats such as XLSX would have been a difficult and tedious task requiring other open-source libraries.

2.3. Dynamic Typing

Tabulate aims to be a user-friendly DSL rather than high-performing. Even though high performance is important for

such a software, it is imperative to make the programming interface easy to use so that users who are not skilled programmers can take advantage of the high performance offered by this language. Therefore, the authors primarily focused on making this DSL easy to learn and understand. One of the steps in that direction is *dynamic typing*, which prevents the user from worrying about static typing and casting, which is handled by *Tabulate*. While implementing dynamic typing, however, the authors ran into a lot of issues involving pointers, as they were heavily used, cast, and dereferenced. Further, many parts of the code required explicit runtime checks before use, which would slow down the runtime of the final compiled executable.

3. Compilation Pipeline

The compilation pipeline adopted by the authors is described in this section.

3.1. Lexical Analysis

The lexer for *Tabulate* is implemented using `flex` in C++ mode. It matches the character stream from the source code using regular expressions, and reports a *lexical error* in case the tokens are ill-formed. The lexical analyser passes these tokens to the parser for syntax analysis using the `yy` namespace functions implemented by the parser, which are described in the next section.

3.2. Syntax Analysis

The parser for *Tabulate* is implemented using `bison` in C++ mode. In particular, the parser is LALR(1). The parser defines the tokens it expects from the lexer, and sets the syntax rules that should be followed by *Tabulate*. As with most implementations using these tools, the parser controls the lexer. Further, special options have been used to change how tokens are input. These can be read about in the `bison` manual.

3.3. Semantic Analysis

The semantic analyzer for *Tabulate* is implemented in the parser file itself. In particular, the parser implements an S-Attributed Translation Grammar (SATG). Apart from the usual semantic actions for various checks in *Tabulate*, the authors have also implemented a `tabulate::driver` interface to make the symbol table and other data structures for semantic analysis to the parser, such as global counters and flags for performing semantic checks specific to some constructs.

3.4. Runtime Libraries

Apart from compile-time semantic checks, *Tabulate* also performs runtime semantic checks since it implements dynamic typing. Examples of such checks include struct member access, array out of bounds, and type checking and coercion. Dynamic typing has been implemented using the `any` class that uses pointer casting.

3.5. Code Generation

Tabulate generates an equivalent C++ code for the instructions specified in its own language. The semantic actions for such code generation are present in the parser itself. Extra attributes for grammar non-terminals were introduced for translation purposes. The code generation is linear and does not warrant a second pass. Easier to use C++ filestream objects are used to output the generated code.

3.6. Compilation of Generated Code

Tabulate also compiles the generated C++ code using the GNU G++ compiler. The code generated complies with the C++17 standard. Thus, by simply calling the *Tabulate* compiler, one can generate executable machine code without having to compile the generated C++ code separately.

4. Building and Running

This section provides the reader with instructions to build *Tabulate* from source and run the compiler on any source code written in this DSL. These instructions are also documented here and are present in Source Code 1. These instructions must be run at a Linux terminal window.

```
cd /path/to/codes
mkdir build
cd build
cmake ..
cmake --build . --target install -j8
./tabulate /path/to/source.txt
./a.out # To execute the compiled
 ↪  translated C++ source code.
```

Source Code 1: Instructions to Build and Run *Tabulate*.

These shell instructions show how to build and use the `CMake` project present in the `codes` folder, with the output directory being `codes/build`. The two binaries produced by this process are `tabulate`, the compiler executable, and `libruntime.a`, a static runtime library to translate and run the translated C++ code.