CS3423 - Compilers II

GD

A detailed Overview of the GD Compiler project

Friday 24th November, 2023

Team Number: 10

Name	Roll Number
K Vivek Kumar	CS21BTECH11026
Bhende Adarsh Suresh	CS21BTECH11008
Jarupula Sai Kumar	CS21BTECH11023
Pathlavath Shankar	CS21BTECH11064

Contents

1 Overview of the Problem Statement **1**
1.1 Importance of the Problem 1

2 Difficulties Faced During Implementation **2**

3 Running the Compiler on Test Cases **2**
3.1 Using Scripting Language 2
3.1.1 Dependencies 2
3.1.2 Running on a Specific Test Case 2

4 Pipeline Working on the Script **3**
4.1 Windows System 3
4.1.1 Script 3
4.1.2 Explanation 3
4.2 Linux System 4
4.2.1 Explanation 5
4.2.2 `plotter.cpp` 5

1. Overview of the Problem Statement

The challenge at hand involves the development of a Dimensional Geometric Domain-Specific Programming language with a C syntax, coupled with integrated plotting capabilities using Python's matplotlib library. This unique programming paradigm introduces users to the world of 2D programming, providing an intuitive environment with familiar C-like syntax. The incorporation of plotting functionalities serves as an effective learning aid, making it easier for individuals to grasp programming concepts.

The project draws inspiration from the concept of buffering, where pre-built data components known as buffer datatypes are utilized to seamlessly store plotting requirements. This approach enhances the understanding of core C++ programming principles, making the language accessible and user-friendly.

1.1 Importance of the Problem

The significance of addressing this problem lies in several key areas:

1. **Educational Impact:** The project serves as an invaluable educational tool, offering a practical and hands-on approach to learning programming. With a syntax resembling C and integrated plotting features, it provides an ideal platform for beginners to gain a solid understanding of programming concepts.

2. **Bridging Theory and Practice:** By introducing buffer datatypes for storing plotting requirements, the project effectively bridges the gap between theoretical knowledge and practical implementation. Users can explore C++ programming concepts while gaining insights into efficient data storage mechanisms.

3. **Enhanced Learning Experience:** Integration with Python's matplotlib library allows users to visualize the real-time impact of their code. This visual context enhances the learning experience, making abstract concepts more tangible.

4. **Versatile Applications:** A Dimensional Geometric Domain-Specific Programming language has diverse applications. From educational purposes to scientific and engineering simulations, users can leverage this language for a wide range of projects, fostering creativity and innovation.

5. **Foundation for Further Exploration:** Proficiency in this language serves as a solid foundation for users to explore more complex programming languages and environments. The project acts as a stepping stone, empowering individuals for future endeavors in the dynamic field of programming and computational science.

Traditional methods of tackling geometry and graph-based problems often involve manual calculations and translations into conventional programming languages, which can be cumbersome. The GD language aims to provide a more intuitive and efficient way to express and solve such problems.

2. Difficulties Faced During Implementation

The implementation of the GD compiler has posed several challenges. Some of the key difficulties encountered include:

- Syntax Design: We wanted a simpler syntax but also wanted to keep it closer to C, hence we resembled ourselves to C++ and doing such resemblance and uniqueness caused in many issues like efficient translation.

  Issues involved in building switch and if-else statement syntax was the most difficult part which took approximately 40% of the project time.

- Environment Handling: Implementing the environment declarations for different dimensions, such as 1D and 2D, required careful consideration.
- Optimizing Operations: Ensuring efficient translation of geometric concepts into executable instructions while optimizing each operation for performance.
- After numerous iterations, we successfully developed the syntax for assigning attributes and allocating a new function table row. This process posed a significant challenge, particularly because it involved delving into Yacc at a profound level for the first time. Addressing character arrays, structs, and array counts proved to be intricate tasks during this exploration.
- Another significant hurdle we faced was in the implementation of array calls and counts. This posed challenges in terms of managing function call stacks and ensuring proper execution flow.

3. Running the Compiler on Test Cases

3.1 Using Scripting Language

For a streamlined end-to-end compilation and execution process, a pre-written script is provided to compile and run the test cases. Ensure that the following dependencies are installed before proceeding:

3.1.1 Dependencies

- `bison:` The Yacc Compiler
- `flex:` The Lex Compiler
- `gcc:` The C Program Compiler
- `g++:` The C++ Compiler
- `python or python3:` Please adjust the command in the `lexer.l` file in the codes directory if you are using python3 instead of python. Navigate to line 67 of `lexer.l` and replace "python" with "python3."
- `matplotlib library from python or python3:` Required for graph generation.

3.1.2 Running on a Specific Test Case

To run the program on a particular test case, follow these steps:

1. Download and unzip the file.
2. Open the directory containing the files.
3. Navigate to the `codes` directory.
4. Run any test case present in the `testcase` folder with the following commands:

   For Windows:

   ```
   $ .\gd.bat ../testcases/t4.txt
   ```

   For Linux Systems:

   ```
   $ chmod +x gd.sh
   $ ./gd.sh input.txt
   ```

4. Pipeline Working on the Script

4.1 Windows System

The following script runs the compiler on Windows:

4.1.1 Script

```
@echo off
bison -d parser.y
if errorlevel 1 (
    echo Error: Bison compilation failed
    exit /b 1
)

flex lexer.l
if errorlevel 1 (
    echo Error: Flex compilation failed
    exit /b 1
)

gcc lex.yy.c parser.tab.c -o vwake
if errorlevel 1 (
    echo Error: Compilation failed
    exit /b 1
)

.\vwake %1
if errorlevel 1 (
    echo Error: Execution failed
    exit /b 1
)

g++ -c svas.cpp -o svas.o
if errorlevel 1 (
    echo Error: Compilation of svas.cpp failed
    exit /b 1
)

g++ -c plotter.cpp -o svas_plotter.o
if errorlevel 1 (
    echo Error: Compilation of plotter.cpp failed
    exit /b 1
)

g++ svas_plotter.o svas.o -o svas
if errorlevel 1 (
    echo Error: Linking failed! No Executable Generated.
    exit /b 1
)

.\svas
if errorlevel 1 (
    echo Error: Execution of svas failed
    exit /b 1
)
```

4.1.2 Explanation

The script follows a step-by-step process to compile and run the code on a Windows system:

- It starts with the Bison compiler to generate the parser code, followed by the Flex compiler for the lexer code.

- The generated codes are then compiled using the GNU Compiler Collection (GCC) to create the `vwake` executable.
- The `vwake` executable is executed with specified test case inputs, and errors are handled appropriately.
- The script then proceeds to compile `svas.cpp` and `plotter.cpp` files separately, linking them to create the `svas` executable.
- Finally, the compiled `svas` executable is run.

4.2 Linux System

The following script runs the compiler on Linux:

```
#!/bin/bash

bison -d parser.y
if [ $? -ne 0 ]; then
    echo "Error: Bison compilation failed"
    exit 1
fi

flex lexer.l
if [ $? -ne 0 ]; then
    echo "Error: Flex compilation failed"
    exit 1
fi

gcc lex.yy.c parser.tab.c -o vwake
if [ $? -ne 0 ]; then
    echo "Error: Compilation failed"
    exit 1
fi

./vwake "$1"
if [ $? -ne 0 ]; then
    echo "Error: Execution failed"
    exit 1
fi

g++ -c svas.cpp -o svas.o
if [ $? -ne 0 ]; then
    echo "Error: Compilation of svas.cpp failed"
    exit 1
fi

g++ -c plotter.cpp -o svas_plotter.o
if [ $? -ne 0 ]; then
    echo "Error: Compilation of plotter.cpp failed"
    exit 1
fi

g++ svas_plotter.o svas.o -o svas
if [ $? -ne 0 ]; then
    echo "Error: Linking failed! No Executable Generated."
    exit 1
fi

./svas
if [ $? -ne 0 ]; then
    echo "Error: Execution of svas failed"
    exit 1
fi

exit 0
```

4.2.1 Explanation

The script for Linux follows a similar process as the Windows script:

- It uses Bison and Flex to generate parser and lexer code, respectively.
- It compiles the generated codes to create the `vwake` executable.
- The `vwake` executable is executed with the provided command-line arguments.
- Similar to the Windows script, the script then compiles `svas.cpp` and `plotter.cpp` files separately, linking them to create the `svas` executable.
- Finally, the compiled `svas` executable is run.

4.2.2 `plotter.cpp`

Compile the `plotter.cpp` file, which contains the implementation of the plotting functionality. The plotter is responsible for interpreting graphical instructions generated by the main program and producing visual representations of the data specified in the input code.

The plotter leverages the Matplotlib library to create graphical elements, including points, lines, and shapes, on a coordinate system.

The compilation of `plotter.cpp` results in an object file (`svas_plotter.o`), which will be later linked with other object files to create the final executable.

Note: Ensure that the necessary libraries are available and properly configured for the compilation of the `plotter.cpp` file.