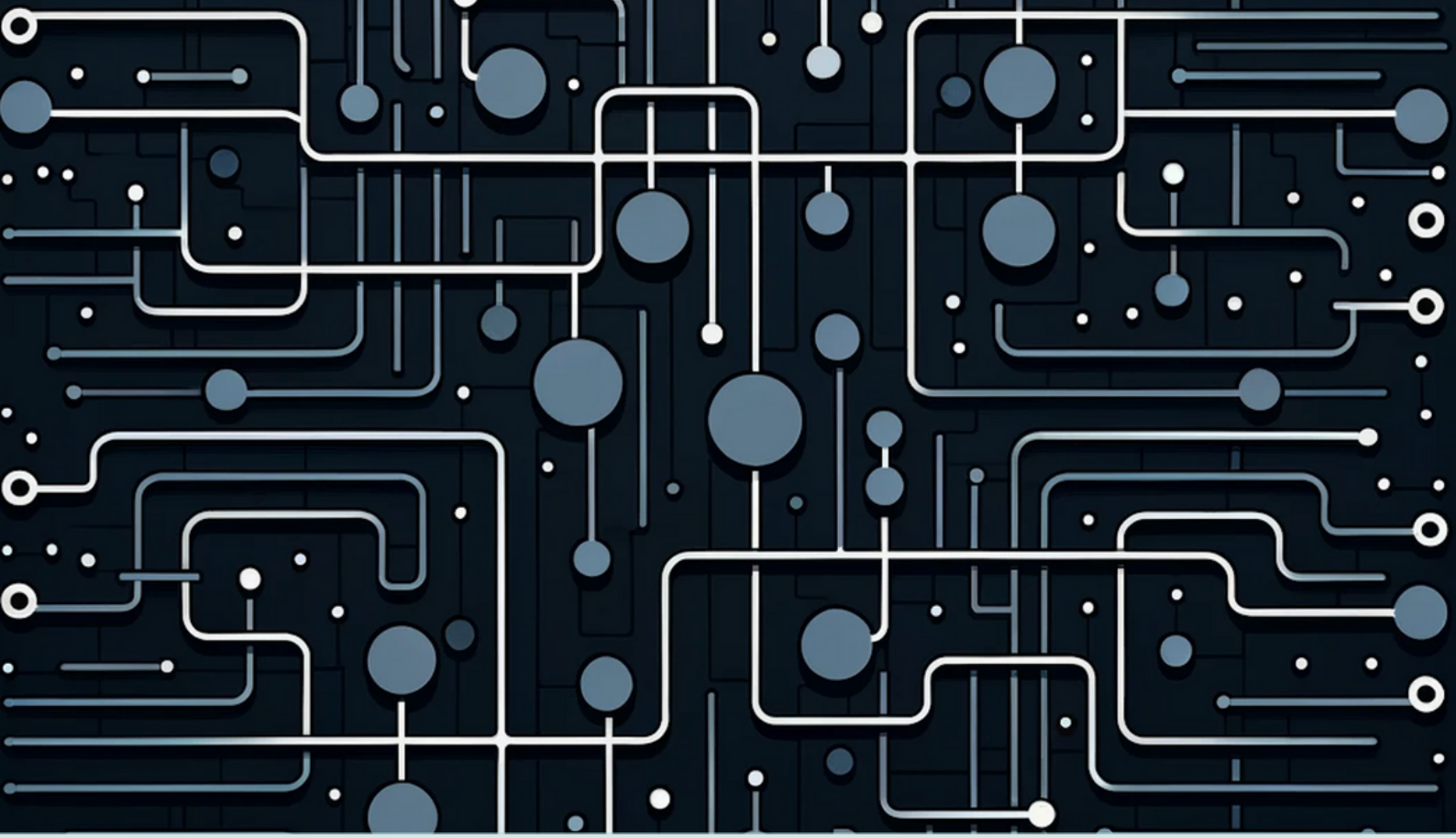
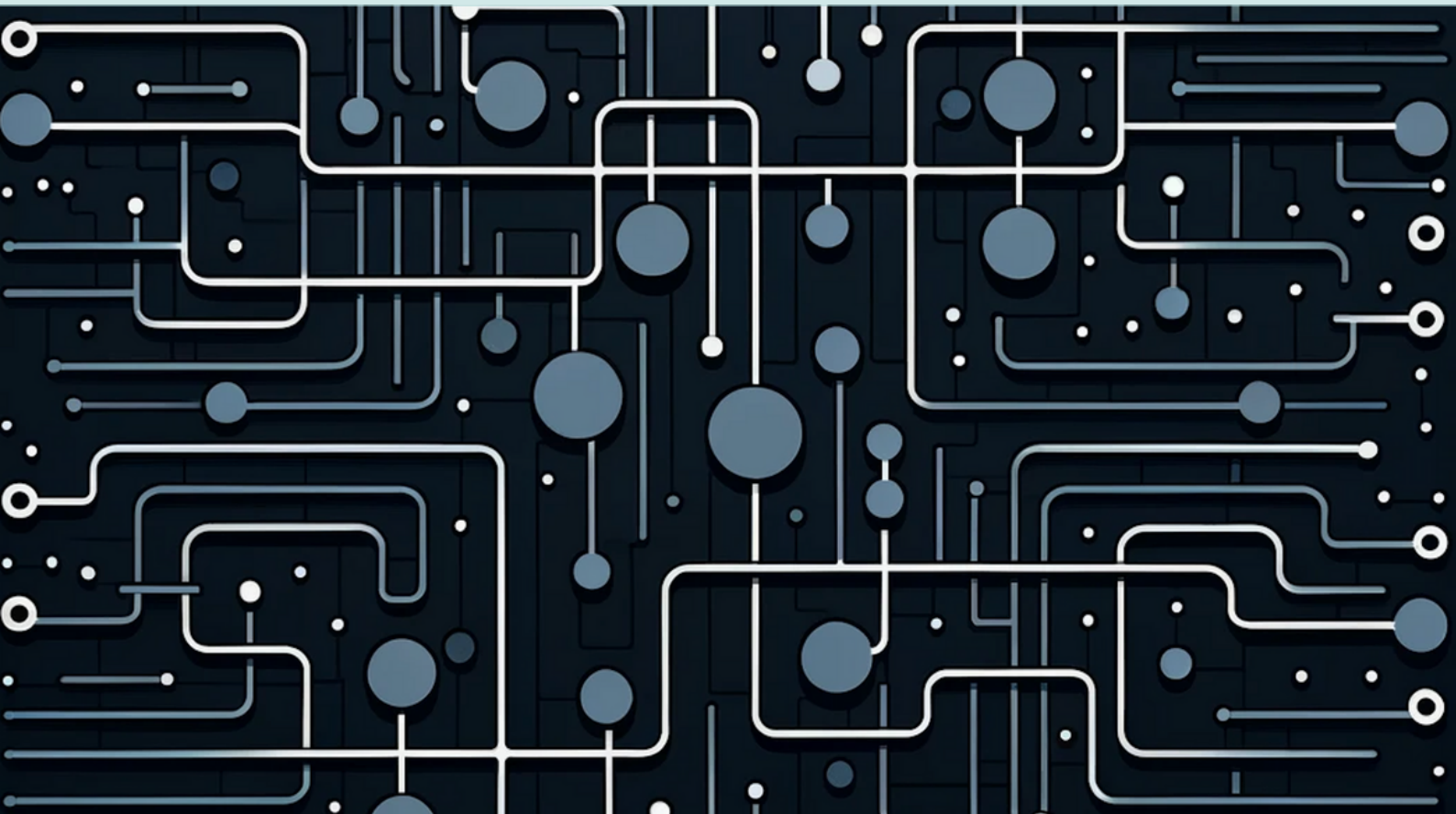


Final Report

Team 15

N. Sai Vamsi Perala Anudeep Rao Varshini Jonnala J. Tushita Sharva



Contents

Introduction

1. About the Project

1.1 Goal

We aimed to design a language simplifies parallel and concurrent software programming. The programmers will have flexibility to work with both lock-based and lock-free data structures.

Additionally, our language places a strong emphasis on **performance analysis**. The integrated tools and constructs of this language enable developers to measure and analyze thread performance, execution times, and program efficiency. This helps one to identify bottlenecks and areas for improvement, ensuring that their parallel code runs efficiently.

Furthermore, the language implements **comprehensive logging capabilities**. Efficient logging mechanisms have been implemented using lock-free queues. This feature helps developers track and analyze program behavior, making it easier to diagnose issues and monitor the execution of parallel code.

The language also encourages the development of parallel algorithms and the creation of parallel versions of common algorithms.

1.2 Importance

Simplified Parallel Programming: The language simplifies the complex parallel programming, making it easier for developers to write efficient code that takes advantage of modern computing capabilities.

Flexible Data Structure Usage: The language offers the choice between lock-based and lock-free data structures, allowing developers to adapt their approach based on specific application needs. This flexibility is crucial for optimizing program efficiency.

Performance Analysis Made Easy: The language integrates performance analysis tools, enabling developers to measure and analyze thread performance, execution times, and overall program efficiency. This aids in identifying and addressing bottlenecks for improved performance.

2. Challenges (& Learning Outcomes)

2.1 Technical Challenges:

- **Collaborative Coding:** Coordinating code development among team members was challenging, we had dealt with version control issues, merge conflicts, and ensuring that everyone is working on the latest version and all. Out of this, we learned how to use git efficiently.
- **Dependency Management:** Managing dependencies between different modules was another minor challenge. Though the work was *distributed*, sometimes, we could not work *concurrently*, since some of the work were co-dependent. We were able to get through this easily by planning meets more often, and having a common workspace, and taking note of each other's work. Even when done, Integrating individual components developed by different team members had sometimes lead to unexpected conflicts.
- **Technology Stack Compatibility:** Ensuring that all team members are using compatible technologies, libraries, and tools is crucial. We helped each other to setup the tools and libraries needed, promoting collaborative learning.

2.2 Non-Technical Challenges:

- **Team Member Availability and Commitment:** Workload distribution and unexpected absences can impact project progress. Ensuring that all team members are equally engaged was crucial.
- **Adapting Our Approach:** Even though we had good brainstorming and discussions, there were times when we got stuck and had to delete that part of our work and start over. For example, we had first come up with an idea with which we could finish the semantic checks part and the code generation part in one go. This approach hadn't worked, at least while the deadline was fast approaching. So, we went back, started afresh again with another idea, and moved forward. We're happy we tried a different approaches.

3. Compiler Flow

1. **Lexical Analysis (`lex.l`):** A tool that generates lexical analyzers. The `lex.l` file contains the rules for tokenizing the input code. It processes these rules and generates `lex.yy.c`, which defines the `yylex()` function.
2. **Syntax Analysis (`bison.y`):** Bison takes a grammar file `bison.y`, which defines the syntax of the language, and produces `bison.tab.c`. This file contains the `yyparse()` function, which uses the tokens from `yylex()` to check if the code follows the correct syntax and builds a parse tree.
3. **Header Files (`bison.tab.h`):** Bison also generates a header file `bison.tab.h` that includes definitions to be used, allowing them to communicate.
4. **Symbol Table (`symbol_table.cpp` and `.hpp`):** The symbol table is a data structure used throughout the compilation process to store information about identifiers (variables, functions, etc.).
5. **Semantic Analysis (`semantics.cpp` and `.hpp`):** The semantic analysis phase uses the symbol table and the parse tree to check for semantic errors, such as type mismatches or undeclared variables. If an error is found, control is passed to the error handling module (`errors.cpp` and `.hpp`).
6. **Error Handling (`errors.cpp` and `.hpp`):** This module likely contains functions for reporting and handling errors encountered during the semantic analysis.
7. **Intermediate Code Generation (`CodeGen.cpp` and `.hpp`):** Once the code is lexically, syntactically, and semantically correct, the `CodeGen` module will translate the parse tree into an intermediate representation. This is a lower-level form of the code that is closer to machine language but is still platform-independent.
8. **Main Driver (`main.cpp`):** This is the central file that drives the compilation process. It calls the `yyparse()` function and likely orchestrates the different phases of the compilation, including invoking the code generation after successful parsing.
9. **Executable (`executable.ex.o`):** The result of the compilation process is an executable file, represented here as `ex.o`. This file contains the machine code that can be executed by the computer.

.cpp file
(READY TO EXECUTE)
input (.cx program)
executable (ex.o)
token & parsed outputs
errors (.cpp and .hpp)
if the look-up fails, send control to error
generates
main.cpp
Calls yyparse( )
CodeGen (.cpp and .hpp)
semantics (.cpp and .hpp)
bison.tab.c has yyparse( );
uses this & subroutines in bison file
lex.yy.c has yylex( );
Calls yylex( )
uses
adds and delete values from and to the table
looks up to verify if code is semantically correct
generates
generates
generates
symbol_table (.cpp and .hpp) Has symbol table
bison.tab.h
bison.y
lex.l

4. Steps to use compiler

Navigate to `./codes` folder from the parent folder. And then run the below bash scripts with the arguments given:

[testcase_file_name] - 1,2,3,4,5,6,7,8

<testcase_folder_name> : 2,3,4
: 2 - Phase-II-testcases
: 3 - Phase-III-testcases
: 4 - Phase-IV-testcases

For performing Lexing, Parsing and Semantic Analysis:

```
bash bash.sh <testcase_folder_name> <testcase_file_name>
```

For Compiling the C++ code Generated:

```
bash cpp_bash.sh <testcase_folder_name> [testcase_file_name]
```