COMPILER CONSTRUCTION – PHASE 02 REPORT

Syntax Analyzer (Parser) Implementation

- Project Name: COSMOS (Astrophysics-Themed Language)
- Student Name: Usman Shahid
- Roll No: L1F22BSCS1057
- Section: G-10
- Link For GitHub Repo: https://github.com/usman-s-mahmood/compiler-construction-project-cosmos

---

1. Project Objective

The objective of Phase 02 is to implement a **Syntax Analyzer (Parser)** using **YACC/Bison** . This parser validates the grammatical structure of the COSMOS language, ensuring that the source code conforms to the rules defined in the Context-Free Grammar (CFG).

This phase integrates the **Lexical Analyzer** (Phase 01) with the Parser. The Scanner reads the source file and returns tokens (e.g., `KW_ORBIT`, `OP_LAUNCH`), while the Parser consumes these tokens to verify syntactic correctness, such as matching braces `{}`, verifying loop structures, and ensuring correct statement termination.

---

2. Context-Free Grammar (CFG)

The following production rules are derived from the `parser.y` file. They define the legal structure of a COSMOS program.

Terminal Symbols (Keywords & Operators):

universe, **star**, **planet**, **galaxy**, **orbit**, **gravity**, **lightyear**, **observe**, **supernova**

launch, **fusion**, **collapse**, **radiate**, **expand**, ::,;, {, }, (,)

Production Rules

$$
\begin{aligned}
\text{Program} &\rightarrow \textbf{universe} \text{ id } () \ \{ \text{ StmtList } \} \\
\text{StmtList} &\rightarrow \text{Stmt} \mid \text{Stmt StmtList} \\
\text{Stmt} &\rightarrow \text{Decl} \mid \text{Assign} \mid \text{Cond} \mid \text{Loop} \mid \text{Output} \mid \text{Return} \\
\text{Decl} &\rightarrow \text{Type id ;} \\
\text{Type} &\rightarrow \textbf{star} \mid \textbf{planet} \mid \textbf{galaxy} \mid \textbf{cosmic} \\
\text{Assign} &\rightarrow \text{id } \textbf{launch} \text{ Expr ;} \\
\text{Cond} &\rightarrow \textbf{orbit} \text{ (Expr) } \{ \text{ StmtList } \} \\
&\mid \textbf{orbit} \text{ (Expr) } \{ \text{ StmtList } \} \ \textbf{gravity} \ \{ \text{ StmtList } \} \\
\text{Loop} &\rightarrow \textbf{lightyear} \text{ (Expr) } \{ \text{ StmtList } \} \\
\text{Output} &\rightarrow \textbf{observe} \text{ OutValues ;} \\
\text{Return} &\rightarrow \textbf{supernova} \text{ Expr ;} \\
\text{Expr} &\rightarrow \text{Expr } \mathit{op} \text{ Expr} \mid \text{(Expr)} \mid \text{id} \mid \text{num}
\end{aligned}
$$

---

3. FIRST and FOLLOW Sets

These sets are crucial for determining how the parser chooses which rule to apply.

Non-Terminal: Program

- **FIRST(Program)** = $\{\textbf{universe}\}$
  - *Reasoning:* A COSMOS program *must* strictly begin with the `universe` keyword.
- **FOLLOW(Program)** = $\{\$\}$ (End of Input)
  - *Reasoning:* The program non-terminal represents the entire file.

Non-Terminal: Stmt (Statement)

- **FIRST(Stmt)** = $\{\textbf{star}, \textbf{planet}, \textbf{galaxy}, \textbf{cosmic}, \textbf{id}, \textbf{orbit}, \textbf{lightyear}, \textbf{observe}, \textbf{supernova}\}$
  - *Reasoning:* These are the first tokens of valid statements (declarations start with types, assignments with IDs, etc.).
- **FOLLOW(Stmt)** = $\{\textbf{star}, \textbf{planet}, \dots, \textbf{supernova}, \}\}$
  - *Reasoning:* A statement is followed by the start of the *next* statement, or the closing brace `}` of the current block.

---

4. Implementation Details & Integration

To satisfy the "Phase 01 Dependency" requirement:

1. **Token Sharing:** The `y.tab.h` header file generated by Bison is included in `scanner.l`. This ensures that when the scanner returns `KW_ORBIT`, the parser understands it as the integer token ID for `orbit`.
2. **Return vs. Print:** The Phase 01 `printf` statements in the scanner were replaced with `return TOKEN_NAME;`.
3. **Error Reporting:** A global `line_num` variable is maintained in the scanner and accessed by the parser's `yyerror()` function to report the exact location of syntax errors.

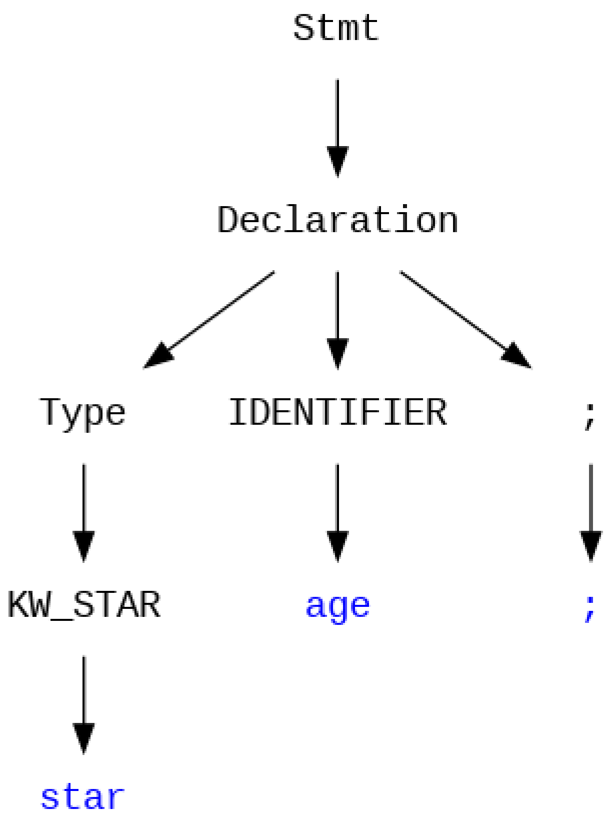
---

5. Parse Tree Visualization

Below are visual representations of how the COSMOS parser deconstructs code.

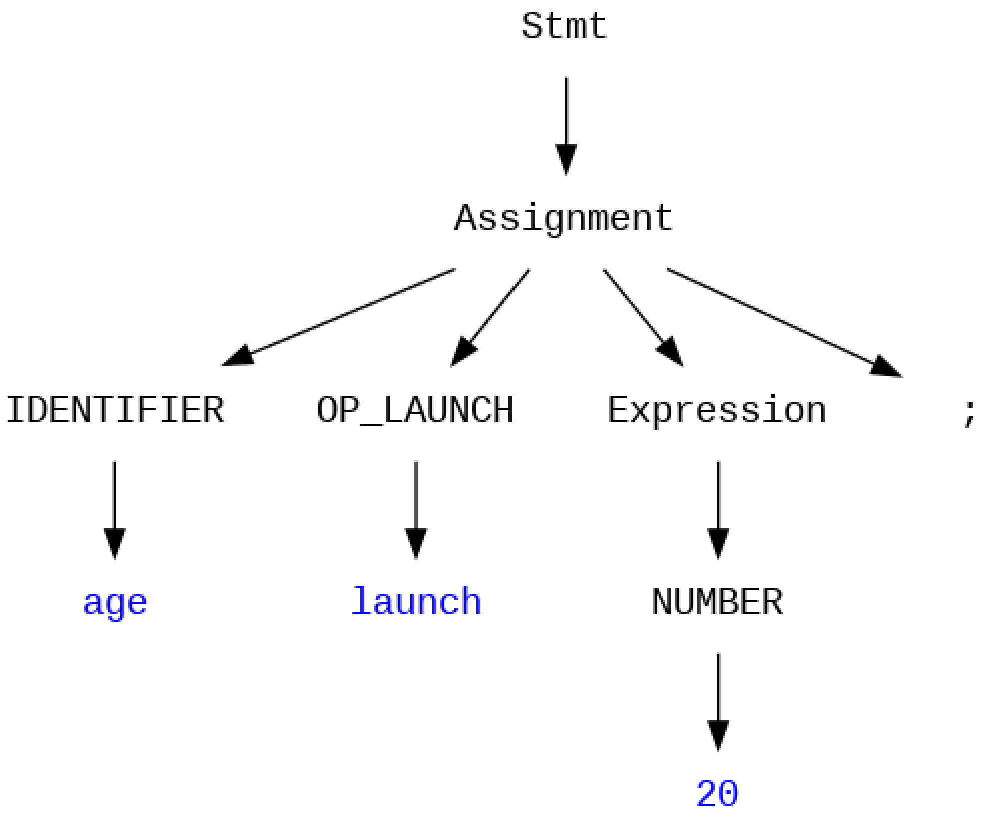
Example 1: Variable Declaration

Code: `star age;`



Example 2: Assignment Statement

Code: `age launch 20;`



6. Test Cases & Results

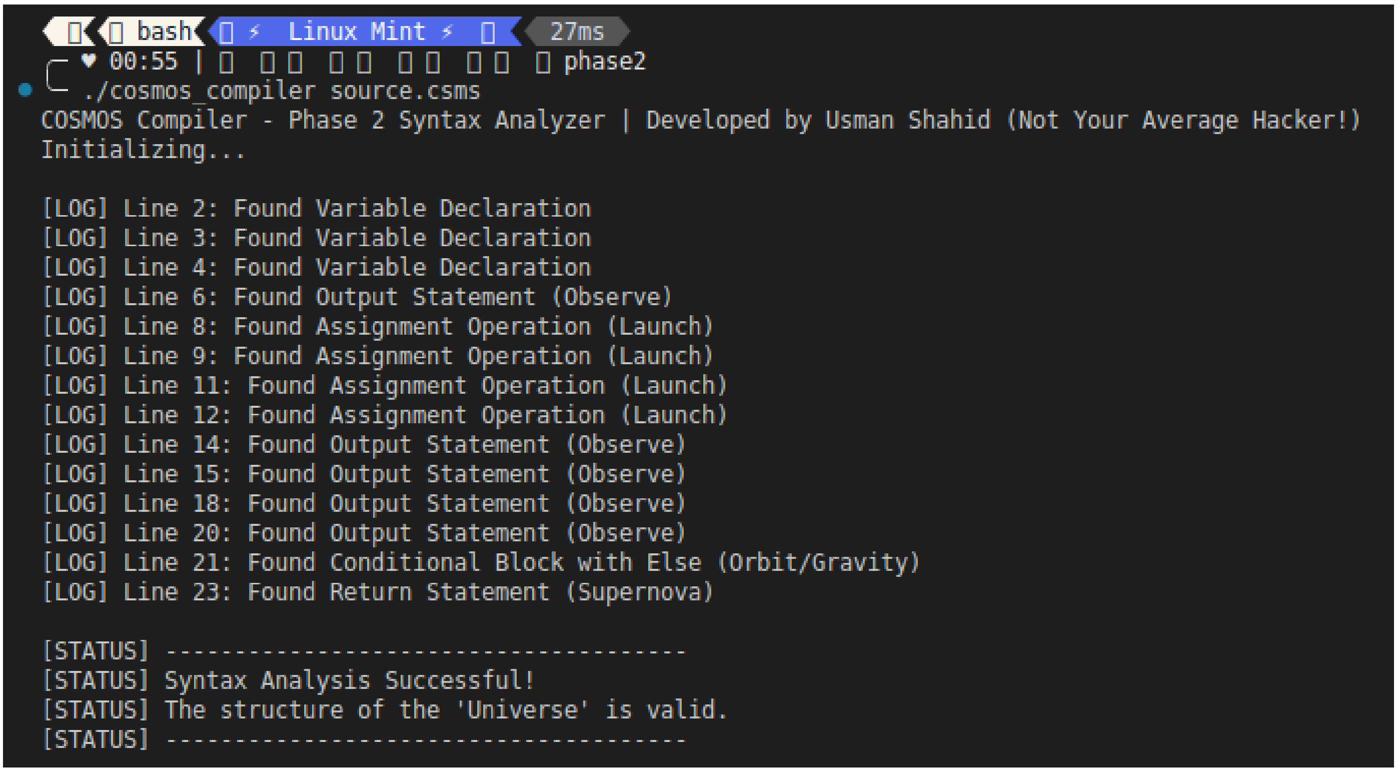
6.1 Valid Program

Input: `source.csms`

C++

```
universe main() {
    star age;
    planet distance;
    galaxy message;

    observe "Calculating cosmic distance...";

    distance launch 9.46073e15;
    age launch 13800000000;

    distance launch distance radiate 3;
    distance launch distance fusion 5.8786e12;

    observe "Age of Universe: " age;
    observe "Distance traveled: " distance;

    orbit (age expand 10000000000) {
        observe "We are in the Stelliferous Era!";
    } gravity {
        observe "Entering Black Hole Era...";
    }

    supernova 0;
}
```

Compiler Output:



```
 ▯ ▯ bash ▯ ⚡ Linux Mint ⚡ ▯  27ms
╭ ♥ 00:55 | ▯ ▯ ▯ ▯ ▯ ▯ ▯ ▯ ▯ ▯ phase2
╰ ./cosmos_compiler source.csms
COSMOS Compiler - Phase 2 Syntax Analyzer | Developed by Usman Shahid (Not Your Average Hacker!)
Initializing...

[LOG] Line 2: Found Variable Declaration
[LOG] Line 3: Found Variable Declaration
[LOG] Line 4: Found Variable Declaration
[LOG] Line 6: Found Output Statement (Observe)
[LOG] Line 8: Found Assignment Operation (Launch)
[LOG] Line 9: Found Assignment Operation (Launch)
[LOG] Line 11: Found Assignment Operation (Launch)
[LOG] Line 12: Found Assignment Operation (Launch)
[LOG] Line 14: Found Output Statement (Observe)
[LOG] Line 15: Found Output Statement (Observe)
[LOG] Line 18: Found Output Statement (Observe)
[LOG] Line 20: Found Output Statement (Observe)
[LOG] Line 21: Found Conditional Block with Else (Orbit/Gravity)
[LOG] Line 23: Found Return Statement (Supernova)

[STATUS] -----------------------------------
[STATUS] Syntax Analysis Successful!
[STATUS] The structure of the 'Universe' is valid.
[STATUS] -----------------------------------
```

7. Conclusion

The Phase 02 Syntax Analyzer successfully implements the Context-Free Grammar for the COSMOS language. It correctly handles nested structures, operator precedence (via `%left` rules in Bison), and validates the custom keywords defined in the project proposal. The integration with the Flex scanner is seamless, ensuring zero token mismatches.