---

# 📖 Chapter 1: The Core Architecture

Before looking at the code, you must understand **how** these two tools talk to each other.

## 1.1 The Two Phases

Your compiler does not read "sentences"; it reads characters. It needs two distinct steps to understand them:

1. **Flex (The Lexer):** This is the **Scanner**. It looks at the raw text stream (C, S, E, 1, 0, 1) and groups them into meaningful blocks called **Tokens** (e.g., COURSE_CODE).
   - *Analogy:* It's like reading a sentence and identifying "Noun", "Verb", "Adjective".
2. **Bison (The Parser):** This is the **Grammar Checker**. It takes the list of tokens from Flex and checks if they follow the rules of your language. If they do, it executes the C code you attached to those rules.
   - *Analogy:* It checks if "Noun + Verb" makes a valid sentence structure.

---

# 📖 Chapter 2: The Lexer (transcript.l)

*The job of this file is to recognize the vocabulary of your language.*

## 2.1 The Setup (Declarations)

C

```
%option noyywrap
%{
#include "transcript.tab.h" // Connects to Bison's token definitions
%}
```

- **%option noyywrap**: Tells Flex, "When you reach the end of the input file, stop." (Don't look for another file).
- **#include "transcript.tab.h"**: This is crucial. Bison generates this header file. It contains the numeric IDs for tokens like CMD_GPA or NUMBER so Flex knows what ID to return.

## 2.2 The Rules (Regex Logic)

This is where you define what a "word" looks like.

**A. Keywords**

Code snippet

```
"CALC_GPA"    { return CMD_GPA; }
```

- **Logic:** When Flex sees the exact text "CALC_GPA", it stops and sends the token ID CMD_GPA to Bison.

**B. Complex Data extraction (The "Gap" you filled)**

Code snippet

```
[A-Z]{3}[0-9]{3}    { yylval.str = strdup(yytext); return COURSE_CODE; }
```

- **The Regex [A-Z]{3}[0-9]{3}**:
  - [A-Z]{3}: Exactly 3 uppercase letters (e.g., CSE).
  - [0-9]{3}: Exactly 3 digits (e.g., 101).
- **yylval.str = strdup(yytext)**:
  - yytext: The actual text found (e.g., "CSE101").
  - yylval: A special variable used to pass **data** to Bison. Since "CSE101" is a string, we duplicate it (strdup) and send it over.

---

# 📖 Chapter 3: The Parser (transcript.y)

*This is the brain of your compiler. It defines structure and executes logic.*

## 3.1 The C Definition Section (Top)

This section contains the **"Complex Function"**, **"File I/O"**, and **"Loop"** logic required by your

teacher.

**Feature A: The Loop (Grade History)**

C

```
char grade_history[50][5]; // A 2D array to store up to 50 grades strings (like "A+")
int grade_count = 0;       // Keeps track of how many grades we stored
```

- *Why?* We cannot count how many 'A's we have while parsing line-by-line. We must store them first and count them at the end using a loop.

**Feature B: The Complex Function (analyze_performance)**

C

```
const char* analyze_performance(float gpa) { ... }
```

- **Logic:** It takes a float (3.8) and returns a string ("Summa Cum Laude"). This fulfills the requirement for **Complex Logic** (nested if-else blocks).

**Feature C: File Writing (log_to_file)**

C

```
FILE *fptr = fopen("academic_log.txt", "a");
```

- **"a" Mode:** This stands for "Append". It ensures previous data isn't deleted.
- **The Loop Implementation:**
  C
  ```
  for(int i = 0; i < grade_count; i++) {
     if(grade_history[i][0] == 'A') { ... }
  }
  ```

- *Explanation:* This iterates through every grade stored in your session to calculate statistics. This is the explicit **Loop** implementation.

## 3.2 The Bison Definitions

Code snippet

```
%union {
   float fval; // For numbers (GPA, Credits)
   char* str;  // For strings (Names, Course Codes)
}
%token <fval> NUMBER
%token <str> COURSE_CODE
```

- **%union**: In C, a variable usually has one type. But a token can be a number OR a string. The Union allows yylval to hold *either* fval (float) or str (string).
- **%token <str>**: Tells Bison, "When you get a COURSE_CODE, expect it to carry string data."

## 3.3 The Grammar Rules (The "Language")

**Rule 1: The Main Program**

Code snippet

```
program: program statement | /* empty */;
```

- *Recursion:* This defines a "list". A program is a statement followed by more program. This allows you to write multiple commands in one file.

**Rule 2: Calculating GPA (mode_gpa_calculation)**

Code snippet

CMD_GPA STRING_LIT L_BRACE { ...Initialization... } course_list R_BRACE { ...Final Math... }

- **Action 1 (Before the list):** When it sees CALC_GPA, it resets variables (local_points = 0). This ensures calculations don't mix between semesters.
- **Action 2 (After the list):** When it sees }, it does the division (points / credits) and calls log_to_file.

**Rule 3: The Recursive List (course_list)**

Code snippet

course_list: course_list course_line | course_line;

- *Concept:* This is **Left Recursion**. It effectively says: "A list is a list plus one more item." This allows your calculator to handle 1 course or 100 courses dynamically.

**Rule 4: The Line Logic (course_line)**

Code snippet

COURSE_CODE COLON NUMBER COLON GRADE SEMICOLON

- *The Math:*
  C
  ```
  float gp = get_grade_point($5); // Convert "A" to 3.75
  local_points += (gp * $3);      // Weighted Sum
  ```
- *The $3, $5 Syntax:*
  - $1 = COURSE_CODE
  - $3 = NUMBER (Credits)
  - $5 = GRADE
  - Bison automatically pulls the values from the tokens based on their position in the rule.

---

# 📖 Chapter 4: Execution Trace

*What happens when you run CALC_GPA "Fall" { CSE101 : 3.0 : A ; }?*

1. **Lexer** sees CALC_GPA. Matches keyword -> Sends token CMD_GPA.
2. **Lexer** sees "Fall". Matches string regex -> Sends STRING_LIT with value "Fall".
3. **Parser** receives these. It matches the start of mode_gpa_calculation.
4. **Action:** It executes the code block { local_points = 0; }.
5. **Lexer** sees CSE101. Matches regex -> Sends COURSE_CODE.
6. **Lexer** sees 3.0. Matches number regex -> Sends NUMBER.
7. **Lexer** sees A. Matches grade regex -> Sends GRADE.
8. **Parser** sees a full course_line.
   - It calls get_grade_point("A") -> returns 3.75.
   - It updates local_points += 3.75 * 3.0.
   - It stores "A" into grade_history.
9. **Lexer** sees }.
10. **Parser** matches the end of mode_gpa_calculation.
    - It calculates GPA = Points / Credits.
    - It calls analyze_performance(GPA) to get the status.
    - It calls log_to_file, which opens the file, loops through grade_history, and saves everything.

---

# 📖 Summary of Requirements Fulfilled

| Teacher's Requirement | Where it lives | Code Implementation |
|---|---|---|
| **Use Loops** | transcript.y (Line 60) | for(int i = 0; i < grade_count; i++) to count grades. |
| **File Writing-Update** | transcript.y (Line 57) | fopen("academic_log.txt", "a") (Append Mode). |
| **Complex Function** | transcript.y (Line 54) | analyze_performance() determines academic standing. |

You are now ready to explain, defend, and run this compiler with confidence!

11111111111111111111111111111111111111