Introduction

The goal of this phase is to help students become familiar with lexical analyzer and parser generator tools, Flex and Bison.

Submission:

Due Date

Phase 1 is due on **September 18, 2025**, at **11:59 p.m**.

How to Submit

1. All your files must be in **p1** directory of your repository (CSC355_<Davidson-Username>).
2. Add, commit, and push the directory.
3. Make sure you can see your files on GitHub online.

Prerequisite

You will need the following software installed on your machine to work on this and the future phases of the project. If you have not done so already, follow the instructions in `CSC355_Student_Fall2025/Auxiliary/GPL Overview and Installation Guide.pdf` to install the required software.

1. C/C++ compiler (GCC, G++, Clang, Clang++).
2. Bison.
3. Flex.
4. OpenGL.

Starter Files

The **ONLY** files you will edit are `record.l` and `record.y`. You **MUST** not change any other files.

- `record.l`: input for the scanner generator, flex.

- `record.y`: input for the parser generator, bison.
- `error.h/error.cpp`: error reporting class; used all semester to make sure all errors are uniform.
- `parser.h`: wrapper for the Bison-generated y.tab.h include file.
- `parser.cpp`: main program that reads command line arguments, starts the parser, and prints information about the execution.
- `tester.py`: this is a Python script comparing the output from your parser to the expected output.
- `tests/`: directory where you will find all the input test files and expected output saved in files.
- `Makefile`: make utility script (talk to instructor if you don't know how to use make).

How to Setup

1. Sync your forked `CSC356_Student_Fall2025` repository to the original repository.
2. Create **p1** directory under your **Project** directory (this is in your GitHub repository).
3. Copy over all the contents (files and directories) in the course repository's `Project/p1/` to your **p1**/.
4. Change the directory to your **p1**/.
5. Check you have all the files and **tests**/ directory in the your **p1**/ (e.g., `$ls .`).
6. Run `$make` to compile the (incomplete) parser.
7. If you see the `parser` binary executable file generated, you are set to begin working on the first phase.

Description

You are provided with a working parser that reads a collection of record declarations written in a *language* for specifying records. After each record is read, the program prints the record to standard output.

Each record is named (like the variable `int value`; it is named `value`). Each element of the record is also named. Given the following input:

```
record george
{
  name = "george";
  age = 31;
  phone = 5551212;
  height = 5.7;
}
```

Your task is to complete the parser to parse the input and output the following correctly:

```
record george
{

  name = 'george' (string)
  age = 31 (int)
  phone = 5551212 (int)
  height = 5.7 (double)

} 4 fields were declared.

1 record was declared.
```

Program Requirements

1. Add a double declaration to the union. Check how an integer union (`union_int`) is declared in `record.y` (line 68).

2. All tokens (e.g., T_ASSIGN, T_ID, etc.) passed from the lexer to the parser must be declared. For example, `%token T_ASSIGN "="`. See how the tokens are declared in `record.y` (line 80 to 98).

   - Each declared token should be mapped to the appropriate regular expression in `record.l`. See lines 86 to 97 in `record.l`.

3. Count the fields in each record [tests: t001, t002].

4. Add fields of type double constant (e.g., 3.1, .42, 42.) [tests: t003, t004].

5. Add fields of type string constant (e.g., "hello world") [tests: t005, t006].
6. Add fields of type date (e.g., FEB 22, 1745, AUG 11, 1892) [tests: t008].
7. Fix the rule for recognizing identifiers (token class: T_ID) [tests: t007, t009, t010].

Hints

1. Count the fields in each record. See how the number of records is updated in `record.y` (line 117) and printed (lines 107 - 110). The counting happens in the `declaration_list` rule, and the count is printed in the `program` rule. Counting fields works the same way. You will count in the rule that is analogous to the `declaration_list` rule and print the total in the rule that is analogous to the `program` rule. The only changes are in `record.y`.
2. Add fields of type double constant. Modify both `record.y` and `record.l`. Replicate integer handling for doubles. Add a new grammar rule:

```
field:
  T_ID T_ASSIGN T_INT_CONSTANT T_SEMIC
  { cout << " " << *$1 << " = " << $3 << " (int)\n"; }
  |
  T_ID T_ASSIGN T_DOUBLE_CONSTANT T_SEMIC
  { cout ... }
  ;
```

3. Add fields of type string constant (e.g., "hello world"). Similar to adding a double. Also requires a new grammar rule. Remove the " from the matched string.
4. Add fields of type date. Do not create a single token. Break it into several tokens: `T_ID T_ASSIGN T_MONTH T_INT_CONSTANT T_COMMA T_INT_CONSTANT T_SEMIC`. Use a `T_MONTH` token that stores the month string.
5. Fix the identifier rules to support underscores and digits (except the first character cannot be a digit).
6. In `record.y`, you should not change productions for `program` and `declaration_list`.

How to Test Your Program

1. Go to **p1** directory, where you should have all your files for phase 1.
2. Compile your parser (`$make`). You should see a `parser` binary executable file generated.
3. Run your parser with a test file, e.g., `$./parser tests/t001.in`.
4. Compare the output with the expected output, e.g., `tests/t001.out`.
5. To test all your programs, run `tester.py`, e.g., `$python3 tester.py`.
6. There are total of **18** test cases.