Introduction

This is the actual first phase of the GPL project. By now, you should be familiar (although not an expert) with how to write specification files for the *flex* (scanner generator) and *bison* (parser generator).

In this phase, you will handle the *basic* syntax of GPL. The scanner needs to recognize the keywords (i.e., reserved identifiers), special symbols (e.g., semicolon, period, etc.), the operators (`+`, `+=`, etc.), and the types (e.g., `int`, `double`, etc.). Then, the parser should be able to parse *all* the grammars listed in the `grammar` file, regardless of whether the rules execute actions or not.

By completing this phase, you will gain a stronger understanding of how *flex* and *bison* interact to build a functioning interpreter. You will also see how the front end of a compiler or interpreter tokenizes source code and applies grammar rules during parsing.

Submission:

Due Date

Phase 2 is due on **October 9, 2025**, at **11:59 p.m**.

How to Submit

1. All your files must be in **p2** 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 `gpl.l` and `gpl.y`. Do **NOT** change any other files.

- `gpl.l`: Lexer specification file.
- `gpl.y`: Parser specification file.
- `gpl.cpp`: `main()` program for gpl. The C pre-processor is used to customize it for the different phases.
- `Makefile`: Makefile to compile the project (this Makefile works for p3–p7).
- `tokens`: Lists all the tokens in gpl.
- `grammar`: Contains all the rules (also called productions) for gpl.
- `parser.h`: Substitutes for `y.tab.h`. Always include `parser.h` instead of `y.tab.h`. (Must update each time you add a new type to the flex/bison union. Do not forget about this file.)
- `error.h` and `error.cpp`: An error reporting class that ensures your errors match my errors letter for letter (**Never change this file**).
- `gpl_assert.h` and `gpl_assert.cpp`: A standard assert implementation that uses functions so they can be traced by the debugger (**Do not change this file**).
- `tests/`: Directory where you will find all the input test files and the expected output is saved in files.
- `gt`: Bash script to test your gpl (this is a replacement of `tester.py` in `p1`). This file's permission should be set to 644, which is not executable. Change the permission to 700 (`chmod 700 gt`).

Setup

1. Create `p2/` directory in your `Project/` directory.
2. Copy over *all* the files listed under **Starter Files** to your `p2/`.
3. Run `make` to make sure you can compile your `gpl`.
4. Try `gt -all`. You should pass *only* `t000`, which is an empty file.
5. Make an initial commit to your repository.

Description

In this phase, you will write specifications for a lexical analyzer (`gpl.l`) and a parser (`gpl.y`) for GPL.

Lexical Analyzer (`gpl.l`) The analyzer should handle all GPL keywords, *all* special symbols (e.g., `;` and `.`), the operators, and the types (`integers`, `doubles`, `strings`). A complete list of tokens is in `p2/tokens`.

Parser (`gpl.y`) Implement productions for *all* grammar rules listed in `p2/grammar`. The parser should accept any syntactically legal GPL program but perform no actions (i.e., productions have empty `{ }` blocks).

Example GPL program (`tests/t002.gpl`):

```
double x;

on leftarrow
{

    x += 1;
}
```

Expected output (`tests/t002.out`):

```
gpl.cpp::main()
  input file(tests/t002.gpl)
  random seed(42)
  read_keypresses_from_standard_input(false)
  dump_pixels(false)
  symbol_table(false)
  graphics(false)

gpl.cpp::main() Calling yyparse()



gpl.cpp::main() after call to yyparse().

No errors found (parser probably worked correctly).

Graphics is turned off by the Makefile. Program exiting.

gpl.cpp::main() done.
```

Program Requirements

- You **must** declare *all* tokens (listed in `p2/tokens`) in `gpl.y` and define their rules in `gpl.l`.
- Declare and define the following functions to be called for an action when the input string is matched with the rule (see `record.l` file from `p1` to reference how such functions are used). These functions are for handling non-static tokens, which require type casting of the values and match with the appropriate token class (basically, generating a token), to be passed to the parser.

  Complete the following emit functions in `gpl.l`:

```
// This function is to handle the string value wrapped with ""
int emit_str_constant(int token)
int emit_id(int token)
int emit_int(int token)
int emit_double(int token)
```

  - **HINT 1:** In GPL, there are two types of tokens:
    * **Static Tokens:** These tokens are those values that are fixed at compile time, e.g., keywords, operators, symbols, etc.).
    * **Non-Static Tokens:** These tokens are those whose values are not fixed at compile time but are determined dynamically while the program runs. There are **four** non-static tokens: (1) identifiers, (2) integers, (3) doubles, and (4) strings.
  - **HINT 2:** Emit static tokens using `int emit(int token)` (see `p2/tokens`).
  - **HINT 3:** To emit the non-static tokens (e.g., `T_INT_CONSTANT`), you must first cast the values to the appropriate type before the emission.
  - **HINT 4:** The emit functions create a token and pass it to the parser.

- Each time you emit a token, increment `line_count` if the token corresponds to the last string in the line (i.e., a string containing `\n`). `count_lines()` function is responsible for incrementing the `line_count` value when called. Think about where and when we want to call this function.

  Tracking line numbers is useful for debugging. For three tokens—`T_EXIT`, `T_PRINT`, and `T_FORWARD`—return the `line_count` value to the parser by emitting the token via `int emit_with_line_number(int token)`.

- Complete the grammar rule section in `gpl.y`. To start, copy all rules from `p2/grammar` into `gpl.y`. Some rules are **incomplete**; your job is to complete them (**HINT:** Review the test files to identify which syntaxes you must support.)

- Note: In this phase, the grammar may appear odd because rules have no actions (bodies); they merely receive tokens and match the appropriate productions.

How to Test Your Program

1. Compile your gpl using make: `$make`.
2. Make sure you have successfully compiled and have `gpl` binary executable file.
3. Change the mode of `gt` file: `$chmod 700 gt`. `gt` file is a bash script that will run the test using your `gpl` program.
4. Run `gt` script: `$./gt`

- `gt` does not run tests you have already passed. To run **all** the tests: `./gt -all`.
- To run the specific test, e.g., t004: `./gt 4` (you do not need the zeros in front of the test number).

5. If you fail a test, you can use `vimdiff` to see the difference.
   - First create a file with **your** output. For example,
     `$./gpl tests/t001.gpl > my001.out`.
   - Compare your file with the expected output file. For example,
     `$vimdiff tests/t001.out my001.out`.
   - The highlighted parts are where the difference exists.
   - If you want to know more about `vimdiff`, check out this post from freeCodeCamp.
6. There are total of **18** test cases.