Introduction

In this phase, your main task is to implement a **symbol table** for your GPL interpreter.

For a compiler, which translates source code into target machine instructions, it needs to collect information about the source program (e.g., identifiers, types, values, etc.) and maintain it throughout various compilation phases. Since there can be a large number of identifiers in the source, it is inefficient to store each identifier's information separately and pass it between phases. Instead, the compiler uses a special data structure called a **symbol table**, which stores all identifier-related information in one place. During compilation, each phase simply needs to know the location of the symbol table and can access it whenever information about identifiers is required.

Submission:

Due Date

Phase 3 is due on **October 23, 2025**, at **11:59 p.m**.

How to Submit

1. All your files must be in **p3** 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

Do **NOT** change `gpl_type.h` and `gpl_type.cpp` files.

- `gpl_type.h`: Contains enumerated types for types, operators, and error status.
- `gpl_type.cpp`: Provides routines for converting the enumerated types in `gpl_types.h` into strings. This is helpful for debugging and for printing.
- `symbol.h`: A header file for the symbol.
- `symbol.cpp`: A class file for the symbol.
- `symbol_table.h`: A header file for the symbol table.
- `symbol_table.cpp`: A class file for the symbol table.
- `Makefile`: Updated Makefile for this and future phases. Replace the Makefile from Phase 2 to this file.

Setup

1. Create `p3/` directory in your `Project/` directory.
2. Copy over *all* the files listed under **Starter Files** to your `p3/`.
3. Copy *all* **your** `p2` files (except the `tests/` directory, as you will get a new test set) to your `p3/` directory.
    - Before copying over, make sure you do not have auxiliary (e.g., `.o`) files and a generated executable gpl binary file.
    - The simplest way is to clean those files using `make clean` command inside of your `p2/` directory.
4. In the `paser.h`, add the following includes:
    - `#include "gpl_type.h"`

    In the file, there should be an instruction where you can add new includes.

5. Run `make` to make sure you can compile your `gpl`.

6. Make an initial commit to your repository.

Description

As mentioned in **Introduction**, your task in this phase is to introduce a symbol table to your interpreter. The header files for both classes are provided. Your task is to define the function for the function declarations that are needed, i.e., functions that do not already have a definition.

In this phase, you will *only* handle the following types of identifiers: `integer`, `double`, and `string`. If your interpreter is working as expected, it will recognize regular variables and arrays of the mentioned types. For example,

```
// Regular variables
int a;
double x;
string s1;
// Arrays
int nums[2];
double values[3];
string strings[4];
```

Note that none of the declarations in the example above has initialization. In this phase, you will not handle the initialization. All identifiers will be assigned a default initial value.

- For all `int` type variables, the initial value is 42.

```
// Example: int a;
int a = 42
```

- For all `double` type variables, the initial value is 3.14159.

```
// Example: double x;
dobule x = 3.14159
```

- For all `string` type variables, the initial value is "`Hello World`".

```
// Example: string s1;
string s1 = "Hello World"
```

- For an array, every index location must be initialized to the array type's default value.

```
// Example: int nums[2];
int nums[0] = 42
int nums[1] = 42
```

To assign the initial values, you will pass the initial value to `Symbol`'s constructor in `gpl.y` within the action of the grammar rule for variable declaration.

Just like any other compiler, GPL does not allow two variables to have the same identifier. Additionally, arrays must be declared with a valid size; that is, all arrays must have a size of ≥ 1. If the input violates these, issue an error using the `Error::error()` reporting function (see `error.h`).

Program Requirements

- Class `Symbol_table` must be implemented as a **singleton**. You MUST use the STL `unordered_map` to store symbols in the symbol table (the key is a C++ string, and the value is a `Symbol *` (you must use a pointer to a symbol)). This declaration is already done for you. See line 29 in `symbol_table.h` file.

- You must implement class `Symbol` to hold the name, value (stored as a void pointer), and type of each variable.

  - You must use the enumerated type `gpl_type.h` to store the type. If you are unfamiliar with how C++'s `enum` works, check out Enumeration in C++, GeeksforGeeks.

  - For a symbol object to hold the values, it points to a dynamically allocated object for the value using `m_data_void_ptr`.

    ```
    m_data_void_ptr = (void *) new int(initial_value);
    ```

    Additionally, check the class member variables. These member variables must be initialized at the time of `Symbol` object creation.

    HINT: For non-array variables, they do not have size. Initialize `m_size` to `UNDEFINED_SIZE`, which is declared as a constant variable at the top of `Symbol.h`.

- The `print()` function in `Symbol_table` class prints all the entities in `symbol_table`. The function will print the identifiers in alphabetical order — the order of the identifier string, not the type name.

Since the `unordered_map` does not sort the entries, you must sort the symbols alphabetically in the symbol table's print function.

Prints all the elements of the symbol table in the following format (order is the default STL map's order):

```
<type> <name> = <value>
```

For example,

```
int a = 42
double b = 3.14159
int c = 42
string d = "Hello world"
int nums[0] = 42
int nums[1] = 42
int nums[2] = 42
```

Note: add " " when printing string constants. The actual string should NOT contain quotes (your scanner should strip the quotes off of string constants).

- **IMPORTANT**

  For this assignment, the following rule in grammar (In `gpl.y`):

```
variable_declaration:
  simple_type T_ID T_LBRACKET expression T_RBRACKET
```

  Must be changed to:

```
variable_declaration:
  simple_type T_ID T_LBRACKET T_INT_CONSTANT T_RBRACKET
```

  In the next phase, you will implement expressions and need to change this rule back (comment out the current rule so it can be easily restored in the next assignment).

Hint

- Now, your interpreter starts to get larger and more complicated as more files interact with each other to build one program, `gpl`. It is essential to understand each file and how they interact. Take a moment to review the files before you start writing any code. This will greatly help you complete this and future phases.

- Take a look at what a C++ `void` pointer is. void Pointer in C++, GeeksforGeeks
- The current grammar initializes `int`, `double`, and `string` variables using an expression. It also uses an expression for the size of arrays. Building the code for these expressions is complex and will be the main part of the next assignment. For this assignment, give default values to variables (see above) so that you can test your `Symbol` class's `set` and `print` functions.
- Write and test your `Symbol` and `Symbol_table` classes before calling them from actions embedded in your `gpl.y`. In other words, write a standalone program to test your `Symbol` and `Symbol_table` classes. It is easier to test code when Bison is not involved.
- When searching an STL `unordered_map` for a specific key, you cannot use the `[]` operator. You must use the `find()` function. Using `[]` when the element is not in the map will insert it into the map. The `find()` function returns an iterator that points to `map.end()` if the target was not found, or to the matching element if it was found.
- Include many error-checking `assert` statements in your program—it will make debugging much easier. Every time you expect a variable to have a specific value, write an `assert`. Standard asserts are implemented using preprocessor directives, which makes it hard to use `gdb` to break on them. In the source code, an `assert` version using a function call is provided so you can break on asserts. Include `gpl_assert.h` to use this version.
- When printing the `Symbol_table`, you must first sort the symbols alphabetically. The easiest way to do this is to place all symbols into a `vector<Symbol *>` and then call the STL `sort` function. The `sort` function requires a comparison function. Define a function

```
bool compare_symbols(Symbol *a, Symbol *b)
```

  that returns `true` if `a`'s `m_name` is less than `b`'s `m_name`.

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. In the test files, you might see something like `// p4 <filename>`. Ignore it, as it's simply that the numbering has changed.

7. There are total of **25** test cases.