Introduction

In this phase, your main task is to implement a **Expression** to add expression trees in the parse tree.

Submission:

Due Date

Phase 4 is due on **November 6, 2025**, at **11:59 p.m**.

How to Submit

1. All your files must be in **p4** 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.

Important Note

According to the author of the GPL project, the students find this phase most challenging, which I strongly agree with. The students who finish this phase on time usually finish the entire project. Please start early as soon as you complete phase 3.

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.

- `constant.h`: Constant header file (**DO NOT CHANGE**).
- `constant.cpp`: Constant class implementation file (**DO NOT CHANGE**).
- `expression.h`: Expression header file.
- `expression.cpp`: Expression class implementation `.cpp` file.
- `variable.h`: Variable header file.
- `variable.cpp`: Variable class implementation `.cpp` file.
- `expression_grammar`: File holding all the rules used to parse expressions.

Setup

1. Create `p4/` directory in your `Project/` directory.
2. Copy over *all* the files listed under **Starter Files** to your `p3/`.
3. Copy *all* **your** `p3` files (except the `tests/` directory, as you will get a new test set) to your `p4/` 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 `p3/` directory.
4. In the `paser.h`, add the following includes:
    - `class Expression;`
    - `class Variable;`

    In the file, there should be an instruction for adding new includes.
5. Make an initial commit to your repository.

Description

In this phase, you will implement the `Expression` class to integrate expression trees into the parse tree. This involves designing a class that represents a single node within an expression tree. Because a pointer to the root node of an expression tree effectively provides access to the entire tree, the entire expression structure can be stored and managed through a single pointer to its root.

After you have implemented the expression class, add actions to your gpl.y that build an expression tree when an expression is parsed. Specifically, you need to implement actions for all rules that have the following non-terminals as their left-hand side (it will help you design the expression class if you understand the code you will write for these actions):

```
expression
primary_expression
optional_initializer
math_operator
variable
```

*Skip the `variable` rules that have a `T_PERIOD` on the right-hand-side, they will be implemented in p5.

Once your parser is building expression trees, add actions to initialize variables of the following types:

- `integer`
- `double`
- `string`

This is done in the rules that have `optional_initializer` as their left-hand side.
When you are done, you will be able to initialize `int`/`double`/`string` variables to any legal expression:

```
int i = 42;
int j = i * 42;
// integers are automatically cast to doubles
double x = i * j * 1.42 / (55 + 2 * i);
string s = "hello" + " " + "world";
// doubles and integers are automatically cast to string
string t = "x = " + x;
```

For assignment `p3`, you changed the rule for declaring arrays:

```
simple_type T_ID T_LBRACKET expression T_RBRACKET
```

was changed to:

```
simple_type T_ID T_LBRACKET T_INT_CONSTANT T_RBRACKET
```

For `p4`, you need to change it back (replace `T_INT_CONSTANT` with `expression`) and update your action for creating an array so that the expression is evaluated to determine the size of the array.

Program Requirements

In addition to the requirements from `p3`, your interpreter must be able to initialize variables (`integers`, `doubles`, `strings`) using a legal expression containing integers, doubles, and/or strings. You must handle an expression according to the size of an array.

In `p3`, all variables (integer, double, string) were initialized to the constants `42`, `3.14159`, and `"Hello world"`. In this assignment, if an optional initializer is not specified, use the values `0`, `0.0`, and `""`. Since `gpl` arrays can never have initial values, they should always be initialized to these default values.

When an error is discovered in an expression that prevents you from creating an expression node, create a constant integer `Expression` with value = 0. This allows the parse to continue and potentially find additional errors before stopping. For example:

```
// The following example does not contain the necessary type-checking expression
    T_DIVIDE expression
{
    if either $1 or $3 is of type STRING, there is an error in the expression
       // Issue the correct error message to create a placeholder expression
           because we can't create the correct expression.
       $$ = new Expression(0); // In my code this creates a constant int
           expression node w/value = 0
    else
      // create the correct expression
      $$ = new Expression($1, T_DIVIDE, $3);
}
```

Listing 1: Error handling in expression parsing

Constant expressions (e.g., `42`, `1.234`, `"hello"`) are always leaf nodes (both the left and right children are `NULL`). The leaf nodes must hold a pointer to a class `Constant` object (see `constant.h`). Constant expressions should never change.

You should not store any `int`/`double`/`string` values in an expression node. Class `Constant` must be used to store constant values. The expression node has a member variable `m_constant`, which is a pointer that refers to the constant object.

Hint

There are two primary functions of class `Expression`: (1) you need to construct the tree, and (2) you need to evaluate the tree (evaluate the expression the tree represents).

Expression construction is straightforward. You create a new node in the expression tree for each `expression` and `primary_expression` rule in the grammar. Since the parser matches expressions in a bottom-up fashion, you build the tree as you parse it.

Consider the example of the addition rule:

```
expression:
 expression T_PLUS expression
 {
        // check that the type of $1 and $3 are compatible
         $$ = new Expression($1, PLUS, $3);
 }
```

Listing 2: Addition rule in expression parsing

All you have to do is create constructors for `Expression` that can handle all the different configurations of an expression tree and call them in the appropriate actions. The constructor shown in the above example is for binary expressions. It takes an operator (from the enumerated type `Operator_type` in `gpl_type.h`) and pointers to two other expressions. The `expression_grammar` file shows all the rules used to parse expressions.

There are four different flavours of an `Expression` node. It is easiest if you implement one class `Expression` and use member variables to track which flavour of `Expression` it is.

- **Constant (int/double/string)**: Leaf node that holds a pointer to a class `Constant` object. `m_lhs` and `m_rhs` pointers are `NULL`. `m_variable` pointer is `NULL`. `m_type` is `INT/DOUBLE/STRING`. `m_oper` is ignored. Example: `42`, `1.2`, `"hello"`.

- **Binary expression**: Non-leaf node that holds an operator (in `m_oper`) and pointers to two expressions. `m_lhs` points to `e1`, `m_rhs` to `e2`. `m_variable` pointer is `NULL`. `m_type` is derived from `e1`, `e2`, and `m_oper`. Example: `e1 + e2`.
- **Unary expression**: Non-leaf node that holds an operator (in `m_oper`) and a pointer to a single expression. `m_lhs` points to `e`, `m_rhs` is `NULL`. `m_variable` pointer is `NULL`. `m_type` is derived from `e` and `m_oper`. Example: `-e`.
- **Variable**: Leaf node that holds a pointer to a `Variable` object (in `m_variable`). `m_lhs` and `m_rhs` are `NULL`. `m_oper` is ignored. `m_type` is retrieved from `m_variable` (i.e., `m_type = m_variable->get_type()`). Example: `i`.

There are six constructors in the `Expression` class. The parameters `lhs` and `rhs` refer to the left-hand and right-hand sides of an operator, while `e`, `e1`, and `e2` are pointers to `Expression` objects.

Expression evaluation functions recursively evaluate each node in the tree. The algorithm performs a depth-first traversal: when a node's evaluation function is called, it evaluates the left child, then the right child, and then applies its operator. If the node holds a constant, it returns the value from the `Constant` object (e.g., `m_constant.get_int_value()`). If it holds a variable, it evaluates the variable (e.g., `my_variable->get_int_value()`) and returns the value.

Expression Types

Expressions have types, just like symbols. An expression's type is the result type of evaluating the expression. While you can determine it dynamically, it is more reliable to assign the type at parse time in the constructor.

For example, an expression with an integer constant has type `INT`. For binary expressions like `a + b`, if both are integers, the type is `INT`; if one is a double, the result is `DOUBLE`; and if either is a string, the result is `STRING`. Relational operators (e.g., <, >, <=) always return `INT`.

The constructors of `Expression` must handle all combinations of operand types correctly. This process is time-consuming but necessary.

For example:

```
simple_type T_ID optional_initializer
{
   if ($1 == INT) // the type of my simple_type token is Gpl_type
   {
      int initial_value = 0; // 0 is the default value for integers
      // if an initializer was specified
      if ($3 != NULL)
      {
         if ($3->get_type() != INT)
            error -- the initializer is not of the correct type
         else initial_value = $3->eval_int();
      }
      // now a new INT symbol can be created using initial_value and *$2.
    }
   // do other cases here (e.g. $1 == DOUBLE)
}
```

Listing 3: Type checking for optional initializer

Expression Evaluation

Evaluating an integer expression differs from evaluating a double or string expression. It is best to define three separate evaluation functions:

```
int Expression::eval_int()
double Expression::eval_double()
string Expression::eval_string()
```

Listing 4: Evaluation functions by type

Relational operators (e.g., `<`, `>`, `<=`, `>=`) are of type `INT` and should only be handled in `eval_int()`. Casting between types (int to double or string, double to string) is implemented in evaluation functions.

Example of casting an integer expression to a string:

```
string Expression::eval_string()
{
    if (m_type == INT)
    {
        int value = eval_int();
        // convert value into a string and return it
    }
    ...
}
```

Listing 5: Casting integer expression to string

Class Variable

The `Variable` class encapsulates all flavors of variables, simplifying the `Expression` class. It enables clean handling of different variable types and will be essential in future phases.

Example variable types in GPL:

```
a
nums[a + b]
my_rectangle.x // implemented in p6
rectangles[i + 2].y // implemented in p6
```

Listing 6: Variable types in GPL

The `Variable` class provides a consistent interface for type and value access:

```
m_variable->get_type()
m_variable->get_int_value()
m_variable->get_double_value()
m_variable->get_string_value()
```

Listing 7: Variable interface functions

For `p5`, the `Variable` class should store both a pointer to a `Symbol` (for all variables) and a pointer to an `Expression` (only for arrays):

```
i // Symbol pointer points to i's Symbol. Expression pointer is NULL.
nums[i * j] // Symbol pointer points to nums' Symbol. Expression pointer points to
    "i * j".
```

Listing 8: Pointers in Variable class

In later phases (p6), it will also hold a member field name (e.g., my_rectangle.x). You must also provide forward declarations in parser.h before including y.tab.h:

```
class Expression;
class Variable;
```

Listing 9: Forward declarations

For the p4 assignment, you will not be adding an action for the following rules:

```
expression -> expression T_NEAR expression

expression -> expression T_touches expression
```

Listing 10: Variable interface functions

Avoiding Common Mistakes

Do not seed the random number generator. It is seeded in gpl.cpp via a command line argument.

The gpl program's int, double, and string values live in two places: class Constant and class Symbol. If you store any gpl int/double/string values anywhere else, you are heading down a path that will prevent you from finishing the assignment.

Never change an int/double/string value stored in class Constant (i.e., do not modify class Constant). Doing so indicates a misunderstanding of how the expression tree works.

If you have started every assignment you have ever done right before the deadline and completed it on time, come to my office, and I will plead with you to start p5 right away. Every student who fails the class starts p5 late.

Warnings

1. The bulk of this assignment is the creation of the data structure to hold an expression tree. In many gpl statements, expressions are evaluated at run time (when the game is running). Thus, we need a data structure to hold an expression so that we can evaluate it at run time. In this phase, you will use expressions only for variable initialization. Although it would be

possible to avoid building an expression tree and simply compute the expression's value at parse time, you **must not** do that. You must implement and build the expression tree in `p4` and evaluate it to initialize variables. If you do not implement expressions correctly now, you will not be able to complete future phases.

2. A small but important part of this assignment is to develop a class that represents a variable, called `Variable`. Some students eliminate this class by pushing its functionality into `Symbol` or `Expression`. **Do not eliminate the Variable class**; it becomes even more important later in the project.

3. There are many parts to this assignment, and it takes more time than most students anticipate. Students who do not finish `p4` within three days of the deadline usually do not pass the class. Do not rely on using late days for `p4`; it rarely works out.

4. **Do not cache any values in any Expression nodes.** For example:

```
int i = 30;
int j = 12;
int k = i + j;
```

Listing 11: Incorrect value caching in Expression nodes

It may seem tempting to think that 30 + 12 equals 42 and to store that value in the `"i + j"` expression node. While this works for this example and phase, it will fail in future phases. The constants (30 and 12) are stored in `Constant` objects, not in expressions.

When constructing an expression, **do not apply any of the operators**. Doing so caches results, which is incorrect. Operators should only be applied during expression evaluation. If you mistakenly evaluate operators during expression creation, you will only discover the problem in `p7`, when it will be too late to fix.

5. For this phase, expression types are `INT`, `DOUBLE`, and `STRING`. In future phases, additional types will be introduced. Do not assume that expressions will always be one of these three types. Your code should follow this structure:

```
if (cur_type == INT)
    handle INT
else if (cur_type == DOUBLE)
    handle DOUBLE
else if (cur_type == STRING) // don't just use "else" assuming STRING
    handle STRING
```

```
else
    error: illegal type
```

Listing 12: Type handling pattern

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 **246** test cases.