CSE340 Spring 2025 Project 1: A Simple Compiler!

Due: **Thursday, February 20 2025 by 11:59 pm MST**

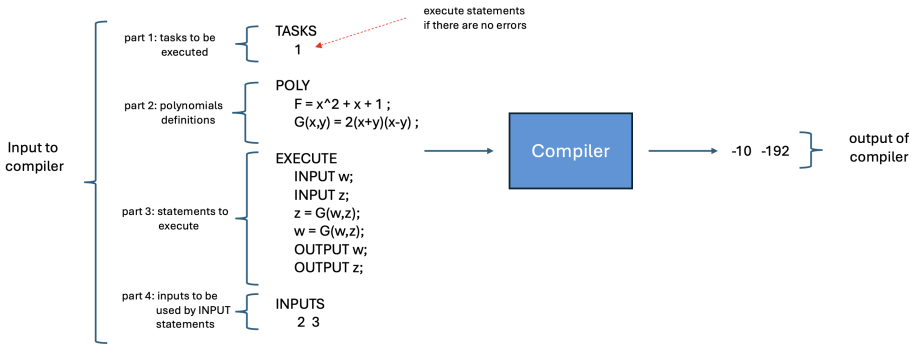
1 Introduction

I will start with a high-level description of the project and its tasks, and in subsequent sections I will give a detailed description on how to achieve these tasks. The goal of this project is to implement a simple compiler for a simple programming language. To implement this simple compiler, you will write a recursive-descent parser and use some simple data structures to implement semantic checking and execute the input program.

The input to your compiler has four parts:

1. The first part of the input is the `TASKS` section. It contains a list of one or more numbers of tasks to be executed by the compiler.
2. The second part of the input is the `POLY` section. It contains a list of polynomial declarations.
3. The third part of the input is the `EXECUTE` section. It contains a sequence of `INPUT`, `OUTPUT` and assignment statements.
4. The fourth part of the input is the `INPUTS` section. It contains a sequence of integers that will be used as the input to `INPUT` statements in the `EXECUTE` section.

Your compiler will parse the input and produces a syntax error message if there is a syntax error. If there is no syntax error, your compiler will analyze semantic errors. If there are no syntax and no semantic errors, your compiler will perform other semantic analyses if so specified by the tasks numbers in the TASKS section. If required, it will *also* execute the `EXECUTE` section and produces the output that should be produced by the `OUTPUT` statements.



The remainder of this document is organized as follows.

- The second section describes the input format.
- The third section describes the expected output when the syntax or semantics are not correct.
- The fourth section describes the output when the program syntax and semantics are correct.
- The fifth section describes the requirements for your solution.

Note: Nothing in this project is inherently hard, but it is larger than other projects that you have done in the past for other classes. The size of the project can make it feel unwieldy. To deal with the size of the project, it is important to have a good idea of what the requirements are. To do so, you should read this document a couple of times. Then, you should have an implementation plan. I make the task easier by providing an implementation guide that addresses some issues that you might encounter in implementing a solution. Once you have a good understanding and a good plan, you can start coding.

2 Input Format

2.1 Grammar and Tokens

The input of your program is specified by the following context-free grammar:

```
program                 →    tasks_section poly_section execute_section inputs_section
tasks_section           →    TASKS num_list
num_list                →    NUM
num_list                →    NUM num_list
poly_section            →    POLY poly_decl_list
poly_dec_list           →    poly_decl
poly_dec_list           →    poly_decl poly_dec_list
poly_decl               →    poly_header EQUAL poly_body SEMICOLON
poly_header             →    poly_name
poly_header             →    poly_name LPAREN id_list RPAREN
id_list                 →    ID
id_list                 →    ID COMMA id_list
poly_name               →    ID
poly_body               →    term_list
term_list               →    term
term_list               →    term add_operator term_list
term                    →    monomial_list
term                    →    coefficient monomial_list
term                    →    coefficient
monomial_list           →    monomial
monomial_list           →    monomial monomial_list
monomial                →    primary
monomial                →    primary exponent
primary                 →    ID
primary                 →    LPAREN term_list RPAREN
exponent                →    POWER NUM
add_operator            →    PLUS
add_operator            →    MINUS
coefficient             →    NUM
execute_section         →    EXECUTE statement_list
statement_list          →    statement
statement_list          →    statement statement_list
statement               →    input_statement
statement               →    output_statement
statement               →    assign_statement
input_statement         →    INPUT ID SEMICOLON
output_statement        →    OUTPUT ID SEMICOLON
assign_statement        →    ID EQUAL poly_evaluation SEMICOLON
poly_evaluation         →    poly_name LPAREN argument_list RPAREN
argument_list           →    argument
argument_list           →    argument COMMA argument_list
argument                →    ID
argument                →    NUM
argument                →    poly_evaluation
inputs_section          →    INPUTS num_list
```

The code that we provided has a class `LexicalAnalyzer` with methods `GetToken()` and `peek()`. Also, an

expect() function is provided. Your parser will use the functions provided to peek()) at tokens or expect() tokens as needed. You must not change these provided functions; you just use them as provided. In fact, when you submit the code, you should not submit the files inputbuf.cc, (inputbuf.h, lexer.cc or lexer.h on gradescope; when you submit the code, the submission site will automatically provide these files, so it is important not to modify these files in your implementation.

To use the provided methods, you should first instantiate a lexer object of the class LexicalAnalyzer and call the methods on this instance. You should only instantiate one lexer object. If you try to instantiate more than one, this will result in errors.

The definition of the tokens is given below for completeness (you can ignore it for the most part if you want).

```
char      =   a | b | ... | z | A | B | ... | Z  | 0 |  1 | ... | 9
letter    =   a | b | ... | z | A | B | ... | Z
pdigit    =   1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
digit     =   0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

SEMICOLON      =  ;
COMMA          =  ,
PLUS           =  +
MINUS          =  -
POWER          =  ^
EQUAL          =  =
LPAREN         =  (
RPAREN         =  )
TASKS          =  (T).(A).(S).(K).(S)
POLY           =  (P).(O).(L).(Y)
EXECUTE        =  (E).(X).(E).(C).(U).(T).(E)
INPUT          =  (I).(N).(P).(U).(T)
OUTPUT         =  (O).(U).(T).(P).(U).(T)
INPUTS         =  (I).(N).(P).(U).(T).(S)
NUM            =  0 | pdigit . digit*
ID             =  letter . char*
```

What you need to do is write a parser to parse the input according to the grammar and produce a syntax error message if there is a syntax error. Your program will also check for semantic errors and, depending on the tasks list, will execute more semantic tasks. To achieve that, your parser will store the program in appropriate data structures that facilitate semantic analysis and allow your compiler to *execute* the statement list in the execute_section. For now, do not worry how that is achieved. I will explain that in detail, partly in this document and more fully in the implementation guide document.

2.2 Examples

The following are examples of input (to your compiler) with corresponding outputs. The output will be explained in more detail in later sections. Each of these examples has task numbers 1 and 2 listed in the tasks_section. They have the following meanings:

- The number 1 listed means that your program should perform syntax and semantic checking.
- The number 2 listed means that your program should produce the output of the output statements if there are no syntax and no semantic errors.

EXAMPLE 1

```
TASKS
   1 2
POLY
```

```
    F = x^2 + 1;
    G = x + 1;
 EXECUTE
   X = F(4);
   Y = G(2);
   OUTPUT X;
   OUTPUT Y;
INPUTS
   1 2 3 18 19
```

This example shows two polynomial declarations and a `EXECUTE` section in which the polynomials are evaluated with arguments 4 and 2 respectively. The output of the program will be

```
17
3
```

The sequence of numbers at the end (in the `input_section`) is ignored because there are no `INPUT` statements.

EXAMPLE 2

```
TASKS
   1 2
POLY
   F = x^2 + 1;
   G = x + 1;
EXECUTE
  INPUT X;
  INPUT Y;
  X = F(X);
  Y = G(Y);
  OUTPUT X;
INPUTS
   1 2 3 18 19
```

This is similar to the previous example, but here we have two `INPUT` statements. The first `INPUT` statement reads a value for `X` from the sequence of numbers and `X` gets the value 1. The second `INPUT` statement reads a value for `Y` which gets the value 2. Here the output will be

```
2
```

Note that the values 3, 18 and 19 are not read and do not affect the execution of the program.

EXAMPLE 3

```
1:    TASKS
2:       1 2
3:    POLY
4:       F = x^2 + 1;
5:       G = x + 1;
6:    EXECUTE
7:      INPUT X;
8:      INPUT Y;
9:      X = F(X);
10:     Y = G(Y);
11:     OUTPUT X;
12:  INPUTS
13:     1 2 3 18 19
```

Note that there are line numbers added to this example. These line numbers are not part of the input and are added only to refer to specific lines of the program. In this example, which looks almost the

same as the previous example, there is a syntax error because there is a missing semicolon on line 4. The output of the program should be

```
SYNTAX ERROR !!!!!&%!!
```

EXAMPLE 4

```
1:    TASKS
2:       1 2
3:    POLY
4:       F = x^2 + 1;
5:       G(X,Y) = X Y^2 + X Y;
6:    EXECUTE
7:      INPUT Z;
8:      INPUT W;
9:      X = F(Z);
10:     Y = G(Z,W);
11:     OUTPUT X;
12:     OUTPUT Y;
12:  INPUTS
13:     1 2 3 18 19
```

In this example, the polynomial `G` has two variables which are given explicitly (in the absence of explicitly named variables, the variable is lower case x by default). The output is

```
2
6
```

EXAMPLE 5

```
1:    TASKS
2:       1 2
3:    POLY
4:       F = x^2 + 1;
5:       G(X,Y) = X Y^2 + X Z;
6:    EXECUTE
7:      INPUT Z;
8:      INPUT W;
9:      X = F(Z);
10:     Y = G(Z,W);
11:     OUTPUT X;
12:     OUTPUT Y;
12:  INPUTS
13:     1 2 3 18 19
```

This example is similar to the previous one but it has a problem. The polynomial `G` is declared with two variables `X` and `Y` but its equation (called `poly_body` in the grammar) has `Z` which is different from `X` and `Y`. The output captures this error (see below for error codes and their format)

```
Semantic Error Code 2: 5
```

3 Tasks and their priorities

The task numbers specify what your program should do with the input program. Task 1 is one of the larger tasks and, but it is not graded as one big task. Task 1 has the following functionalities:

1. Syntax checking
2. Semantic error checkings

The other tasks, 2, 3, 4, 5 and 6 have the following functionalities:

- **Task 2 – Output**: Task 2 requires your compiler to produce the output that should be produced by the output statements of the program. .
- **Task 3 – Variable used but not explicitly initialized**: Task 3 requires your compiler to produce a warning about uninitialized variables. A variable is uninitialized when a variable appears on the right-hand side of an assignment statement without having previously appeared on the left-hand side of an assignment statement or in an `INPUT` statement. This will result in a warning message. However, it is not considered a semantic error. The execution can proceed assuming the variable is initially zero.
- **Task 4 – Useless assignments**: This happens when a variable value is calculated, but the variable is not used later in the right-hand side of an assignment or in an `OUTPUT` statement.
- **Task 5 – Polynomial degree**: This task requires that the degree of all the polynomials in the polynomial sections are calculated and outputted.

Detailed descriptions of these tasks and what the output should be for each of them is given in the sections that follow. The remainder of this section explains what the output of your program should be when multiple task numbers are listed in the `tasks_section`.

If task 1 is listed in the `tasks_section`, then task 1 should be executed. Remember that task 1 performs syntax error checking and semantic error checking. If the execution of task 1 results in an error, and task 1 is listed in the `tasks_section`, then your program should only output the error messages (as described below) and exits. If task 1 results in an error (syntax or semantic) no other tasks will be executed even if they are listed in the `tasks_section`. If task 1 is listed in the `tasks_section` and does not result in an error message, then task 1 produces no output. In that case, the outputs of the other tasks that are listed in `tasks_section` should be produced by the program. The order of these outputs should be according to the task numbers. So, first the output of task 2 is produced (if task 2 is listed in `tasks_section`), then the output of task 3 is produced (if task 3 is listed in `tasks_section`) and so on.

If task 1 is not listed in the `tasks_section`, task 1 still needs to be executed. If task 1's execution results in an error, then your program should output nothing in this case. If task 1 is not listed and task 1's execution does not result in an error, then the outputs of the other tasks that are listed in `tasks_section` should be produced by the program. The order of these outputs should be according to the task numbers. So, first the output of task 2 is produced, then the output of task 3 is produced (if task 3 is listed in `tasks_section`) and so on.

You should keep in mind that tasks are not necessarily listed in order in the `tasks_section` and they can even be repeated. For instance, we can have the following TASKS section:

```
TASKS
      1 3 4 1 2 3
```

In this example, some tasks are listed more than once. Later occurrences are ignored. So, the `tasks_section` above is equivalent to

```
TASKS
      1 2 3 4
```

In the implementation guide, I explain a simple way to read the list and sort the task numbers using a boolean array.

4 Task 1 – Syntax and Semantic Checking

For task 1, your solution should detect syntax and semantic errors in the input program as specified in this section.

4.1 Syntax Checking

If the input is not correct syntactically, your program should output

```
SYNTAX ERROR !!!!!&%!!
```

If there is syntax error, the output of your program should exactly match the output given above. No other output should be produced in this case, and your program should exit after producing the syntax error message. The provided `parser.*` skeleton files already have a function that produces the message above and exits the program.

4.2 Semantic Checking

Semantic checking also checks for invalid input. Unlike syntax checking, semantic checking requires knowledge of the specific lexemes and does not simply look at the input as a sequence of tokens (token types). I start by explaining the rules for semantic checking. I also provide some examples to illustrate these rules.

- **Polynomial declared more than once – Semantic Error Code 1**. If the same `polynomial_name` is used in two or more different `polynomial_header`'s, then we have the error *polynomial declared more than once*. The output in this case should be of the form

  ```
  Semantic Error Code 1: <line no 1> <line no 2> ... <line no k>
  ```

  where `<line no 1>` through `<line no k>` are the numbers of each of the lines in which a duplicate `polynomial_name` appears in a polynomial header. The numbers should be sorted from smallest to largest. For example, if the input is (recall that line numbers are not part of the input and are just for reference):

  ```
  1:    TASKS
  2:       1 3 4
  3:    POLY
  4:       F1 =
  5:             x^2 + 1;
  6:       F2 = x^2 + 1;
  7:       F1 = x^2 + 1;
  8:       F3 = x^2 + 1;
  9:       G = x^2 + 1;
  10:      F1 = x^2 + 1;
  11:      G(X,Y) = X Y^2 + X Y;
  12:  EXECUTE
  13:      INPUT Z;
  14:      INPUT W;
  15:      X = F1(Z);
  16:      Y = G(W);
  17:      OUTPUT X;
  18:      OUTPUT Y;
  19:  INPUTS
  20:      1 2 3 18 19
  ```

  then the output should be

  ```
  Semantic Error Code 1: 7 10 11
  ```

  because on each of these lines the name of the polynomial in question has a duplicate declaration. Note that only the line numbers for the duplicates are listed. The line number for the first occurrence of a name is not listed.

- **Invalid monomial name – Semantic Error Code 2**. There are two kinds of polynomials headers. In the first kind, only the polynomial name (`ID`) is given and no parameter list (`id_list` in the header) is given. In the second kinds, the header has the form `polynomial_name LPAREN id_list RPAREN`. In a polynomial with the first kind of header, the polynomial should be univariate (one variable) and the variable name should be lower case "x". In a polynomials with the second kind of header, the `id_list` is the list variables that can appear in the polynomial body. An `ID` that appears in the body of a polynomial (in `primary`) should be equal to one of the variables of the polynomial. If that is not the case, we say that we have an *invalid monomial name error* and the output in this case should be of the form:

```
Semantic Error Code 2: <line no 1> <line no 2> ... <line no k>
```

where `<line no 1>` through `<line no k>` are the numbers of lines in which an invalid monomial name appears with one number printed per occurrence of an invalid monomial name. If there are multiple occurrences of an invalid monomial name on a line, the line number should be printed multiple times. The line numbers should be sorted from smallest to largest.

- **Attempted evaluation of undeclared polynomial – Semantic Error Code 3**. If there is no polynomial declaration with polynomial name which is the same as a polynomial name used in a polynomial evaluation, then we have *attempted evaluation of undeclared polynomial error*. In this case, the output should be of the form

```
Semantic Error Code 3: <line no 1> <line no 2> ... <line no k>
```

where `<line no 1>` through `<line no k>` are the numbers of each of the lines in which a `polynomial_name` appears in a `polynomial_evaluation` but for which there is no `polynomial_declaration` with the same name. The line numbers should be listed from the smallest to the largest. For example if the input is:

```
1:    TASKS
2:       1 3 4
3:    POLY
4:       F1 = x^2 + 1;
5:       F2 = x^2 + 1;
6:       F3 = x^2 + 1;
7:       F4 = x^2 + 1;
8:       G1 = x^2 + 1;
9:       F5 = x^2 + 1;
10:      G2(X,Y) = X Y^2 + X Y;
11:   EXECUTE
12:      INPUT Z;
13:      INPUT W;
14:      X = G(Z);
15:      Y = G2(Z,W);
16:      X = F(Z);
17:      Y = G2(Z,W);
18:   INPUTS
19:      1 2 3 18 19
```

then the output should be

```
Semantic Error Code 3: 14 16
```

Because on line 14, there is an evaluation of polynomial `G` but there is no declaration for polynomial `G` and on line 16, there is an evaluation of polynomial `F` but there is no declaration of polynomial `F`.

- **Wrong number of arguments – Semantic Error Code 4**. If the number of arguments in a polynomial evaluation is different from the number of parameters in the polynomial declaration, then we say that we have *wrong number of arguments error* and the output should be of the form:

```
Semantic Error Code 4: <line no 1> <line no 2> ... <line no k>
```

where `<line no 1>` through `<line no k>` are the numbers of each of the lines in which `polynomial_name` appears in a `polynomial_evaluation` but the number of arguments in the polynomial evaluation is different from the number of parameters in the corresponding polynomial declaration. The line numbers should be listed from the smallest to the largest. For example if the input is:

```
1:    TASKS
2:       1 3 4
3:    POLY
4:       F1 = x^2 + 1;
```

```
5:       F2 = x^2 + 1;
6:       F3 = x^2 + 1;
7:       F4 = x^2 + 1;
8:       G1 = x^2 + 1;
9:       F5 = x^2 + 1;
10:      G2(X,Y) = X Y^2 + X Y;
11:  EXECUTE
12:      INPUT Z;
13:      INPUT W;
14:      X = G2(X,Y, Z);
15:      Y = G2(Z,W);
16:      X = F1(Z);
17:      Y = F5(Z,Z);
18:      Y = F5(Z,Z,W);
19:  INPUTS
20:      1 2 3 18 19
```

then the output should be

```
Semantic Error Code 4: 14 17 18
```

You can assume that an input program will have only one kind of semantic errors. So, for example, if a test case has Semantic Error Code 2, it will not have any other kind of semantic errors.

5 Task 2 – Program Output

For task 2, your program should output the results of all the polynomial evaluations in the propram. In this section I give a precise definition of the meaning of the input and the output that your compiler should generate. In a separate document that I will upload a little later, I will give an implementation guide that will help you plan your solution. You do not need to wait for the implementation guide to write the parser!

5.1 Variables and Locations

The program uses names to refer to variables in the `EXECUTE` section. For each variable name, we associate a unique locations that will hold the value of the variable. This association between a variable name and its location is assumed to be implemented with a function `location` that takes a `string` as input and returns an integer value. We assume that there is a variable `mem` which is an array with each entry corresponding to one variable. **All variables should be initialized to 0 (zero)**.

To allocate `mem` entries to variables, you can have a simple table or map (which I will call the *location table*) that associates a variable name with a location. As your parser parses the input program, if it encounters a variable name in an `input_statement`, it needs to determine if this name has been previously encountered or not by looking it up in the location table. If the name is a new variable name, a new location needs to be associated with it, and the mapping from the variable name to the location needs to be added to the location table. To associate a location with a variable, you can simply keep a counter that tells you how many locations have been used (associated with variable names). Initially, the counter is 0. The first variable will have location 0 associated with it (will be stored in `mem`[0]), and the counter is incremented to become 1. The next variable will have location 1 associated with it (will be stored in `mem`[1]), and the counter is incremented to become 2 and so on.

For example, if the input program is

```
1:    TASKS
2:       1 2
3:    POLY
4:       F1 = x^2 + 1;
5:       F2(x,y,z) = x^2 + y + z + 1;
6:       F3(y) = y^2 + 1;
7:       F4(x,y) = x^2 + y^2;
```

```
8:       G1 = x^2 + 1;
9:       F5 = x^2 + 1;
10:      G2(X,Y,Z,W) = X Y^2 + X Z + W + 1;
11:  EXECUTE
12:      INPUT X;
13:      INPUT Z;
14:      Y = F1(Z);
15:      W = F2(X,Z,Z);
16:      OUTPUT W;
17:      OUTPUT Y;
18:      INPUT X;
19:      INPUT Y;
20:      INPUT Z;
21:      Y = F3(X);
22:      W = F4(X,Y);
23:      OUTPUT W;
24:      OUTPUT Y;
25:      INPUT X;
26:      INPUT Z;
27:      INPUT W;
28:      W =  G2(X,Z,W,
29:              Z);
30:  INPUTS
31:      1 2 3 18 19 22 33 12 11 16
```

Then the locations of variables will be

```
X 0
Z 1
Y 2
W 3
```

5.2 Statements

We explain the semantics of the four kinds of statements in the program.

5.2.1 Input statements

Input statements get their input from the sequence of `inputs`. We refer to i'th value that appears in `inputs` as `i'th input`. The i'th input statement in the program of the form `INPUT X` is equivalent to:

```
mem[location("X")] = i'th input
```

5.2.2 Output statements

Output statements have the form `OUTPUT ID` where the lexeme of the token `ID` is a variable name. This is the *output variable* of the output statement. Output statements print the values of their OUTPUT variables. If the output statement has the form `OUTPUT X;` , its effect is equivalent to:

```
cout << mem[location("X")]  << endl;
```

Note that each output statement produces its output on a separate line.

5.2.3 Assignment statements

Assignment statements have the form:

```
ID EQUAL poly_evaluation SEMICOLON
```

If the lexeme of the `ID` token of an `assign_statement` is "X", and the `poly_evaluation` of the `assign_statement` has value `v` (see below for the value of a polynomial evaluation), then the assign statement execution will have an effect equivalent to

```
mem[location("X")]  = v;
```

5.2.4 Polynomial Evaluation

The polynomial evaluation depends on the evaluation of arguments and the correspondence between the arguments in polynomial evaluation and the parameters in the polynomial declaration.

Argument Evaluation. The *value* of a variable `X` at a given point in the program is equal to the last value assigned to `X` before that point (a variable is assigned a value either in an `input_statement` statement or in an `assign_statement`) . If there is no prior assignment to `X` before that point, then the value of `X` is 0 (zero). The definition of what an argument evaluates to depends on the definition of what a polynomial evaluates to because an argument can be a polynomial evaluation. An argument is evaluated as follows:

- If the argument is an `ID` whose lexeme is "X", then it evaluates to the value of "X" at that point.
- If the argument is a polynomial evaluation, then it evaluates to what the polynomial evaluates to (see below).

Correspondence Between Arguments and Parameters. In a polynomial declaration, the list of parameters is given by the `id_list` in the header or if the header has no `id_list`, then the parameter is the unique variable "x" (lower case). In a polynomial evaluation, the argument list is given by the `argument_list`. We say that the i'th argument in a polynomial evaluation *corresponds to* the i'th parameter of the polynomial declaration.

Evaluation of a `coefficient`. A coefficient whose lexeme is `L` evaluates to the integer represented by `L`.

Evaluation of an `exponent`. An exponent whose lexeme is `L` evaluates to the integer represented by `L`.

Evaluation of a `monomial`. There are a number of cases to consider

- A monomial of the form `ID` whose lexeme is "X" evaluates to the argument corresponding to "X"
- A monomial of the form `ID exponent` where the lexeme of the `ID` is "X" evaluates to v^e where e is the value that the exponent evaluate to and v is the value that the argument corresponding to "X" evaluates to.

Evaluation of a `monomial_list` . A `monomial_list` of the form `monomial` evaluates to the value that `monomial` evaluates to. A `monomial_list` of the form `monomial monomial_list'` evaluates to the product of v (the value that `monomial` evaluates to) and v' (the value that `monomial_list'` evaluates to).

Evaluation of a `term`. A term of the form `coefficient monomial_list`, where `coefficient` evaluates to c and `monomial_list` evaluates to v, evaluates to $c \times v$ (the product of c and v).

Evaluation of a `term_list`. A `term_list` of the form `term` evaluate to the value that `term` evaluates to. A `term_list` of the form `term add_operator term_list'`, where `term` evaluates to v and `term_list'` evaluates to v', evaluates to $v + v'$ if the `add_operator` is `PLUS` and to $v - v'$ if the `add_operator` is `MINUS`.

Evaluation of a `polynomial_body`. A `polynomial_body` of the form `term_list` evaluates to the value that the `term_list` evaluates to.

Evaluation of a polynomial. A polynomial evaluates to the value that its `polynomial_body` evaluates to.

5.3 Assumptions

You can assume that the following semantic errors are not going to be tested

1. You can assume that if there is a polynomial declaration with a given polynomial name, then there is no variable with the same name in the program.
2. If you want to use an array for the `mem` variable, you can use an array of size 1000 which should be enough for all test cases, but make sure that your code handles overflow (more than 1000 variables in the program) because that is good programming practice.

6 Task 3 – Variable used before being initialized

Uninitialized argument. Warning Code 1. If an `argument` in an `argument_list` in a `polynomial_evaluation` does not appear in an `input_statement` nor on the lefthand side of an assignment statement before the polynomial evaluation, then we say that we have an *uninitialized argument warning* and the output should be of the form:

```
Warning Code 1: <line no 1> <line no 2> ... <line no k>
```

where `<line no 1>` through `<line no k>` are the numbers of each of the lines in which an `argument` appears in a `polynomial_evaluation` but for which: (1) which there is no previous `assign_statement` in which the argument appears on the lefthand side of the `EQUAL` sign and (2) there is no previous `input_statement` in which the `argument` appears. The line numbers should be listed from the smallest to the largest. For example, if the input is:

```
1:    TASKS
2:       1 2 3
3:    POLY
4:       F1 = x^2 + 1;
5:       F2(x,y,z) = x^2 + y + z + 1;
6:       F3(y) = y^2 + 1;
7:       F4(x,y) = x^2 + y^2;
8:       G1 = x^2 + 1;
9:       F5 = x^2 + 1;
10:      G2(X,Y,Z,W) = X Y^2 + X Z + W + 1;
11:  EXECUTE
12:      INPUT X;
13:      INPUT Z;
14:      Y = F1(Z);
15:      W = F2(W,Z, W);
16:      INPUT Y;
17:      INPUT Z;
18:      Y = F4(X,Q);
19:      W = F4(X,Y);
20:      W = F2(W, P, P);
21:      OUTPUT Y;
22:      INPUT X;
23:      W =  G2(X,Z,W,
24:              Z);
25:  INPUTS
26:      5 2 3 18 19 22 33 12 11 16
```

then the output will be

```
25
Warning Code 1:  15 15 18 20 20
```

We first note that there is a `25` printed before the warning message. This is the output of the program (produced in line 21). Notice how the calculation is performed using value 5 for `X` (which is read from INPUTS

on line 12) and value 0 for `Q` (initial value 0 for Q because Q is not initialized). The output of the program should be produced in this case because there are no syntax and no semantic errors, and 2 appears in the TASKS section, so Task 2 should be executed.

Now, we explain the warning message. Notice that line 15 is repeated in the output because both `W`'s on line 15 are not initialized before they appear in the polynomial evaluation `F2(W,Z,W)`. Line 18 is also listed in the warning message because in line 18, Q is used in the evaluation of `F4(X,Q)` without previously being initialized. Line 20 is repeated in the output because `P` which is used twice in the calculation of `F2(W,P,P)` without being previously initialized. Finally, note that there is no warning for lines 23 and 24 because all arguments are previously initialized: `X` on line 22, `Z` on line 17 and `W` on line 20.

7 Task 4 – Useless assignments

A useless assignment occurs if a variable is assigned a value in an assignment statement, but the variable is not used later on the righthand side of an assignment in an `assign_statement` or in an `output_statement`. The definition requires some explanation. Consider the following program:

```
1:   TASKS
2:       1 4
3:   POLY
4:       F = 2x + 1;
5:   EXECUTE
6:       X = F(5);
7:       X = F(6);
8:       OUTPUT X;
9:       X = F(7);
10:      X = F(X);
11:      INPUT X;
12:      OUTPUT X;
13:   INPUTS
14:      2 4 6
```

We examine each assignment statement in the program.

> `Line 6: X = F(5);` The value calculated in this statement is not used later in an `output_statement` or in an `assign_statement` because the next statement on line 7 overwrites the calculated value.
>
> `Line 7: X = F(6);` The value calculated in this statement is used by the output statement that appears on line 8.
>
> `Line 9: X = F(7);` The value calculated in this statement appears on the right-hand side of the `assign_statement` on line 10, so the assignment is not useless.
>
> `Line 10: X = F(X);` The value calculated in this statement is not used later in an `output_statement` or in an `assign_statement` because it is immediately followed by `INPUT X;` which overwrites the value of `X`.

In general, we can define useless assignments as follows. We say that a statement *defines* a variable `x` if it is of the form `INPUT x;` or if it is of the form `x = poly_evaluations ;` We say that a statement *uses* a variable `x` if the statement is of the form `OUTPUT x;` or if the statement is of the form `ID = poly_evaluation ;` and `x` appears in `poly_evaluation` as an argument.

Given the statement list $\texttt{stmt}_1\ \texttt{stmt}_2\ \ldots\ \texttt{stmt}_k$, if $\texttt{stmt}_i$ has the form `x = poly_evaluations ;` , we say that $\texttt{stmt}_i$ is a useless assignment statement if $\texttt{stmt}_i$ is the last statement in the list, or $\texttt{stmt}_{i+1}$ *defines* `x`, or $\texttt{stmt}_{i+1}$ does not *use* `x` and $\texttt{stmt}_i$ is useless in the sequence $\texttt{stmt}_1\ \texttt{stmt}_2\ \ldots\ \texttt{stmt}_i\ \texttt{stmt}_{i+2}\ \ldots\ \texttt{stmt}_k$

If the program has useless assignments, then its output should be of the form:

```
Warning Code 2: <line no 1> <line no 2> ... <line no k>
```

where `<line no 1>` through `<line no k>` are the numbers of each of the lines that contains a useless assignment statement as defined above. The line numbers should be listed from the smallest to the largest, and you can assume that useless assignment statements are not split across multiple lines.

8 Task 5 – Polynomial degree

This tasks requires that the degree of polynomials in the polynomial sections be calculated and outputted. The degree of a polynomial is defined as follows:

1. The degree of a primary of the form `ID` is 1.
2. The degree of a primary of the form `LPAREN term_list RPAREN` is equal to the degree of `term_list`.
3. The degree of a monomial of the form `primary` is equal to the degree of `primary`.
4. The degree of a monomial of the form `primary exponent` is equal to the degree of `primary` times the value of the `exponent`.
5. The value of an exponent of the form `POWER NUM` is equal to the value of `NUM`, which is the integer that corresponds to the lexeme of `NUM`.
6. The degree of a monomial list of the form `monomial` is equal to the degree of the `monomial`.
7. The degree of a monomial list of the form `monomial monomial_list` is equal to the sum of the degree of the `monomial` and the degree of `monomial_list`.
8. The degree of a term of the form `coefficient` is equal to 0 (zero).
9. The degree of a term which is of the form `coefficient monomial_list` is equal to the degree of the `monomial_list`.
10. The degree of a term which is of the form `monomial_list` is equal to the degree of the `monomial_list`.
11. The degree of a term list of the form `term` is equal to the degree of `term`.
12. The degree of a term list of the form `term add_operator term_list` is equal to the maximum of the degree of `term` and the degree of `term_list`.

If there are no syntax and no semantic errors, the output for this task should have the form

```
<poly 1>: <degree 1>
<poly 2>: <degree 2>
<poly 3>: <degree 3>
  ...
<poly k>: <degree k>
```

where `<poly i>` is the name of the i'th polynomial and `<degree i>` is the degree of the i'th polynomial.

For example, if the input is:

```
1:    TASKS
2:       1 2 5
3:    POLY
4:       F1 = x^2x + x^3x^1;
5:       F2(x,y,z) = x^2 + y + z + 1;
6:       F3(y) = y^2 + 1;
7:       F4(x,y) = x^2 + y^2;
8:       G1(x,y,z,w) = x^2 y z w +  x^2 y^2 z w  + 1;
9:       F5 = x^2 + 1;
10:      G2(X,Y,Z,W) = X Y^2 + X Z + W + 1;
11:   EXECUTE
12:      INPUT X;
13:      INPUT Z;
14:      Y = F1(Z);
```

```
15:      W = F2(W,Z, W);
16:      INPUT Y;
17:      INPUT Z;
18:      Y = F4(X,Q);
19:      W = F4(X,Y);
20:      W = F2(W, P, P);
21:      OUTPUT Y;
22:      INPUT X;
23:      W =  G2(X,Z,W,
24:              Z);
25:  INPUTS
26:      5 2 3 18 19 22 33 12 11 16
```

The output will be:

```
25
F1: 4
F2: 2
F3: 2
F4: 2
G1: 6
F5: 2
G2: 3
```

By now, you should know why the output has 25 on the first line.

9 Requirements

You should write a program to generate the correct output for a given input as described above. You should start by writing the parser and make sure that it correctly parses the input before attempting to implement the rest of the project.

You will be provided with a number of example test cases. These test cases are not meant to be complete or even close to complete. They are only provided as examples to complement the project description. It is still your responsibility to make sure that your implementation satisfies the requirements given in this document, and you should develop your own test cases to do so.

10 Instructions

Follow these steps:

- Read this document carefully.
- When the implementation guide is posted, read it carefully. It has detailed explanations on how to approach the implementation.
- Download the `lexer.cc`, `lexer.h`, `inputbuf.cc` and `inputbuf.h` files accompanying this project description and familiarize yourself with the provided functions.
- Design a solution before you start coding. It is really very important to have a clear overall picture of what the project entails before you start coding. Deciding on data structures and how you will use them is crucial. One possible exception is the parser, which you can and should write first before the rest of the solution.
- Write your code and make sure to compile your code using GCC (4:11.2.0) on **Ubuntu 22.04** (Ubuntu). If you want to test your code on your personal machine, you should install a virtual machine with Ubuntu 22.04 and the correct version of GCC on it. You will need to use the `g++` command to compile your code in a terminal window. See section 12 for more details on how to compile using GCC. **You are required to compile and test your code on Ubuntu using the GCC compiler**, but you are

free to use any IDE or text editor on any platform while developing your code as long as you compile it and test it on Ubuntu/GCC before submitting it.

- Test your code to see if it passes the provided test cases. You will need to extract the test cases from the zip file and run the provided test script `test1.sh`. See section 12 for more details.
- Develop your own test cases and test your program on them.
- Submit your code through gradescope.
- You are allowed an unlimited number of submissions, but it is your responsibility to activate the submission that should count for your grade. If have a submission with a grade of 90 and another submission with a grade of 80. You should make sure to activate the older submission if you want it to count. Activating older submissions is also allowed during the late submission period. You can continue on working to improve your grade, but you can revert to an earlier grade if you want by activating an earlier submission.

Keep in mind that

- You should use C++11, no other programming languages or versions of C++ are allowed.
- **All programming assignments in this course are individual assignments. Students must complete the assignments on their own.**
- You should submit your code through gradescope; no other mode of submission will be accepted.
- You should familiarize yourself with the Ubuntu environment and the GCC compiler. Programming assignments in this course are different from assignments in other classes.

11 Evaluation

Submissions are evaluated using automated test cases on gradescope. To receive credit for parsing, your submission should pass ALL the parsing test cases. For other categories, your grade will be proportional to the number of test cases that your submission passes.If your code does not compile on gradescope, you will not receive any points even if it runs correctly in a different environment (Windows/Visual Studio for example). Here is the grade breakdown for the various categories:

1. **Task 1 – Parsing: 30%**. There is no partial credit for parsing. Your program should pass **ALL** the parsing test cases to get credit for parsing, otherwise no credit will be given for parsing.
2. **Task 1 – Semantic Error Code 1: 7.5%**. The grade for this category will be proportional to the number of test cases that are correctly handled by your program.
3. **Task 1 – Semantic Error Code 2: 7.5%**. The grade for this category will be proportional to the number of test cases that are correctly handled by your program.
4. **Task 1 – Semantic Error Code 3: 7.5%**. The grade for this category will be proportional to the number of test cases that are correctly handled by your program.
5. **Task 1 – Semantic Error Code 4: 7.5%**. The grade for this category will be proportional to the number of test cases that are correctly handled by your program.
6. **Task 2 – Program Output: 20%**. The grade for this category will be proportional to the number of test cases that are correctly handled by your program.
7. **Task 3 – Variable used before being initialized: 10%**. The grade for this category will be proportional to the number of test cases that are correctly handled by your program.
8. **Task 4 – Useless assignments: 10%**. The grade for this category will be proportional to the number of test cases that are correctly handled by your program.
9. **Task 5 – Polynomial degree: 10%**. The grade for this category will be proportional to the number of test cases that are correctly handled by your program.

12 General instructions for all programming assignments

NOTE: This section applies to all programming assignments.

You should use the instructions in the following sections to compile and test your programs for all programming assignments in this course.

12.1 Compiling your code with GCC

You should compile your programs with the GCC compilers. GCC is a collection of compilers for many programming languages. There are separate commands for compiling C and C++ programs. Use the `g++` command to compile C++ programs

Here is an example of how to compile a simple C++ program:

```
$ g++ test_program.cpp
```

If the compilation is successful, it will generate an executable file named `a.out` in the same directory (folder) as the program. You can change the executable file name by using the `-o` option. For example, if you want the executable name to be `hello.out`, you can execute

```
$ g++ test_program.cpp -o hello.out
```

To enable C++11, with `g++`, which you should do for projects in this class, use the `-std=c++11` option:

```
$ g++ -std=c++11 test_program.cpp -o hello.out
```

The following table summarizes some useful compiler options for g++:

Option	Description
`-o path`	Change the filename of the generated artifact
`-g`	Generate debugging information
`-ggdb`	Generate debugging information for use by GDB
`-Wall`	Enable most warning messages
`-std=c++11`	Compile C++ code using 2011 C++ standard

Compiling projects with multiple files

If your program is written in multiple source files that should be linked together, you can compile and link all files together with one command:

```
$ g++ file1.cpp file2.cpp file3.cpp
```

Or you can compile them separately and then link:

```
$ g++ -c file1.cpp
$ g++ -c file2.cpp
$ g++ -c file3.cpp
$ g++ file1.o file2.o file3.o
```

The files with the `.o` extension are object files but are not executable. They are linked together with the last statement (g++ file1.o file2.o file3.o) and the final executable will be `a.out`.

12.2 Testing your code on Ubuntu

Your programs should not explicitly open any file. You can only use the **standard input** and **standard output** in C++. The provided lexical analyzer already reads the input from standard input and you should not modify it. In C++, standard input is `std::cin` and standard output is `std::cout`. In C++, any output that your program produces should be done with `cout`. To read input from a file or produce output to a file, we use IO redirection outside the program. The following illustrates the concept.

Suppose we have an executable program `a.out`, we can run it by issuing the following command in a terminal (the dollar sign is not part of the command):

```
$ ./a.out
```

If the program expects any input, it waits for it to be typed on the keyboard and any output generated by the program will be displayed on the terminal screen.

To get the input to the program from a file, we can redirect the standard input to a file:

```
$ ./a.out < input_data.txt
```

Now, the program will not wait for keyboard input, but rather read its input from the specified file as if the file `input_data.txt` is standard input. We can redirect the output of the program as well:

```
$ ./a.out > output_file.txt
```

In this way, no output will be shown in the terminal window, but rather it will be saved to the specified file[1].

Finally, it's possible to do redirection for standard input and standard output simultaneously. For example,

```
$ ./a.out < input_data.txt > output_file.txt
```

will read standard input from `input_data.txt` and produces standard output to `output_file.txt`.

Now that we know how to use standard IO redirection, we are ready to test the program with test cases.

Test Cases

For a given input to your program, there is an *expected* output which is the correct output that should be produced for the given input. So, a test case is represented by two files:

- `test_name.txt`
- `test_name.txt.expected`

The input is given in `test_name.txt` and the expected output is given in `test_name.txt.expected`.

To test a program against a single test case, first we execute the program with the test input data:

```
$ ./a.out < test_name.txt > program_output.txt
```

With this command, the output generated by the program will be stored in `program_output.txt`. To see if the program generated the correct expected output, we need to compare `program_output.txt` and `test_name.txt.expected`. We do that using the `diff` command which is a command to determine differences between two files:

```
$ diff -Bw program_output.txt test_name.txt.expected
```

If the two files are the same, there should be no difference between them. The options `-Bw` tell `diff` to ignore whitespace differences between the two files. If the files are the same (ignoring the whitespace differences), we should see no output from `diff`, otherwise, `diff` will produce a report showing the differences between the two files.

We consider that the test **passed** if `diff` could not find any differences, otherwise we consider that the test **failed**.

Our grading system uses this method to test your submissions against multiple test cases. In order to avoid having to type the commands shown above for running and comparing outputs for each test case manually, we provide you with a script that automates this process. The script name is `test1.sh`. `test1.sh` will make your life easier by allowing you to test your code against multiple test cases with one command.

Here is how to use `test1.sh` to test your program:

- Store the provided test cases zip file in the same directory as your project source files
- Open a terminal window and navigate to your project directory

[1] Programs have access to another standard stream which is called standard error e.g. `std::cerr` in C++. Any such output is still displayed on the terminal screen. It is possible to redirect standard error to a file as well, but we will not discuss that here

- Unzip the test archive using the `unzip` command: `bash  $ unzip tests.zip`

This will create a directory called `tests`

- Store the `test1.sh` script in your project directory as well
- Make the script executable: `bash  $ chmod +x test1.sh`
- Compile your program. The test script assumes your executable is called `a.out`
- Run the script to test your code: `bash  $ ./test1.sh`

The output of the script should be self explanatory. To test your code after you make changes, you will just perform the last two steps (compile and run `test1.sh`).