CSE340 Fall 2021 Project 3: A Simple Compiler!

Due: **Monday November 8 2021 by 11:59 pm MST**

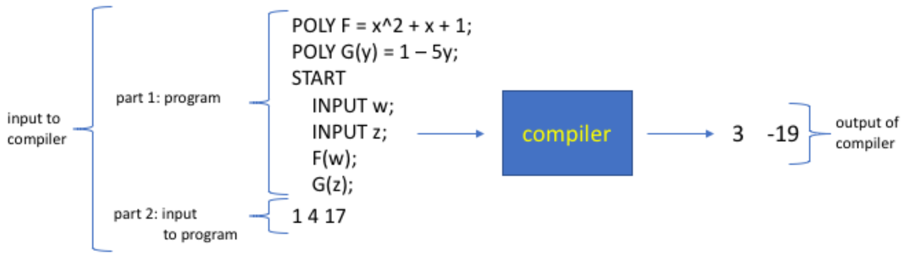
1 Introduction

I will start with a high-level description of the project and its tasks in this section and then, in subsequent sections, I will go into 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. By implementing this simple compiler, you will do basic parsing and use some simple data structures which would be useful for the other projects.

The input to your program will have two parts:

1. The first part of the input is a program which is a list of polynomial declarations followed by a `START` section.
2. The second part of the input is a sequence of integers which will be used as the input to the program given in the first part.

Your compiler will read the program and makes sure that its syntax and semantics are correct. If the syntax or semantics are not correct, the compiler will output an error message as described later. If input program has correct syntax and semantics, the compiler should evaluate the polynomial expressions in the `START` section of the program and the output of the compiler is the sequence of values resulting from the evaluations of the polynomials in the `START` section. More details about the input format and the expected output of your program are given in subsequent sections. The following is a high-level illustration of what your compiler should do for a particular example program whose syntax and semantics are correct.



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 on your solution.

2 Input Format

2.1 Grammar and Tokens

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

```
input                      →    program inputs
program                    →    poly_decl_section start
poly_decl_section          →    poly_decl
poly_decl_section          →    poly_decl poly_decl_section
poly_decl                  →    POLY polynomial_header EQUAL polynomial_body SEMICOLON
polynomial_header          →    polynomial_name
polynomial_header          →    polynomial_name LPAREN id_list RPAREN
id_list                    →    ID
id_list                    →    ID COMMA id_list
polynomial_name            →    ID
polynomial_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                   →    ID
monomial                   →    ID exponent
exponent                   →    POWER NUM
add_operator               →    PLUS
add_operator               →    MINUS
coefficient                →    NUM
start                      →    START statement_list
inputs                     →    NUM
inputs                     →    NUM inputs
statement_list             →    statement
statement_list             →    statement statement_list
statement                  →    input_statement
statement                  →    poly_evaluation_statement
poly_evaluation_statement  →    polynomial_evaluation SEMICOLON
input_statement            →    INPUT ID SEMICOLON
polynomial_evaluation      →    polynomial_name LPAREN argument_list RPAREN
argument_list              →    argument
argument_list              →    argument COMMA argument_list
argument                   →    ID
argument                   →    NUM
argument                   →    polynomial_evaluation
```

The code that we provided has a class `LexicalAnalyzer` with methods `GetToken()`, `peek()` and `UngetToken()`. Your parser will use the provided functions to peek at tokens or get and unget tokens as needed. You must not change the functions textttGetToken() and `peek()` ; you just use them as provided. In fact, when you submit the code, you should not submit the files for the lexical analyzer that we provided (`inputbuf.*` and `lexer.*`) on the submission site; when you submit the code, the submission site will automatically provide these files, so it is important that you do not 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         =  )
POLY           =  (P).(O).(L).(Y)
START          =  (S).(T).(A).(R).(T)
INPUT          =  (I).(N).(P).(U).(T)
NUM            =  0 | pdigit . digit*
ID             =  letter . char*
```

What you need to do is to write a parser to parse the input according to the grammar and store the information being parsed by your parser in appropriate data structures to allow your program to *execute* the input `program` on the `inputs`. For now do not worry how that is achieved. I will explain that in details, partly in this document and more fully in the implementation guide document.

2.2 Examples

The following are examples of inputs (to your compiler) with corresponding outputs. The output will be explained in more details in the later sections.

1.
   ```
   POLY F = x^2 + 1;
   POLY G = x + 1;
   START
   F(4);
   G(2);
   1 2 3 18 19
   ```

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

   ```
   17 3
   ```

   For this example, the sequence of numbers at the end (the input to the program) does not affect the output of the program.

2.
   ```
   POLY F = x^2 + 1;
   POLY G = x + 1;
   START
   INPUT X;
   INPUT Y;
   F(X);
   G(Y);
   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 3
```

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

3.
```
POLY F = x^2 + 1
POLY G = x + 1;
START
INPUT X;
INPUT Y;
F(X);
G(Y);
1 2 3 18 19
```

   In this example, which looks almost the same as the previous example, there is a syntax error because there is a missing semicolon on the first line. The output of the program should be

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

4.
```
POLY F = x^2 + 1;
POLY G(X,Y) = X Y^2 + X Y;
START
INPUT Z;
INPUT W;
F(Z);
G(Z,W);
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
```

5.
```
POLY F = x^2 + 1;
POLY G(X,Y) = X Y^2 + X Z;
START
INPUT Z;
INPUT W;
F(Z);
G(Z,W);
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 `polynomial_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)

```
Error Code 2: 2
```

3 Syntax and Semantic Error Checking

Your solution should detect syntax and semantic errors in the input program as specified in this section.

3.1 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.

3.2 Semantic Checking

Semantic checking also checks for invalid input. Unlike syntax checking, semantic checking requires knowledge of the specific lexemes and does not simply looks at the input as a sequence of tokens (token types). I start by giving the rules for semantic checking and I also give examples for some semantic checking rules. We have the following rules for semantic checking.

- **Polynomial declared more than once. 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

  ```
  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 (the line numbers in the examples below are not part of the input and are just for reference):

  ```
  1: POLY F1 =
  2:            x^2 + 1;
  3: POLY F2 = x^2 + 1;
  4: POLY F1 = x^2 + 1;
  5: POLY F3 = x^2 + 1;
  6: POLY G = x^2 + 1;
  7: POLY F1 = x^2 + 1;
  8: POLY G(X,Y) = X Y^2 + X Y;
  9: START
  10: INPUT Z;
  11: INPUT W;
  12: 1 2 3 18 19
  ```

  then the output should be

  ```
  Error Code 1: 1 4 6 7 8
  ```

  because on each of these lines the name of the polynomial in question has a duplicate declaration.

- **Invalid monomial name. 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. A monomial that appears in the body of a polynomial should have an `ID` that is 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:

  ```
  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. Error Code 3**. If there is no polynomial declaration with the same 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

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

then the output should be

```
Error Code 3: 13 15
```

Because on line 13, there is an evaluation of polynomial `F` but there is no declaration for polynomial `F` and on line 15, there is an evaluation of polynomial `G` but there is no declaration of polynomial `G`.

- **Wrong number of arguments. 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:

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

then the output will be

```
Error Code 4:  15 17 18
```

- **Uninitialized argument. Error Code 5.** If an `argument` in an `argument_list` in a `polynomial_evaluation` does not appear in an `input_statement` before the polynomial evaluation, then we say that we have an *uninitialized argument error* and the output should be of the form:

  ```
  Error Code 5: <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` for which 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: POLY F1 =
  2:            x^2 + 1;
  3: POLY F2(x,y,z) = x^2 + y + z + 1;
  4: POLY F3(y) = y^2 + 1;
  5: POLY F4(x,y) = x^2 + y+ 1;
  6: POLY G1 = x^2 + 1;
  7: POLY F5 = x^2 + 1;
  8: POLY G2(X,Y,Z,W) = X Y^2 + X Z + W + 1;
  9:
  10: START
  11: INPUT Y;
  12: INPUT Z;
  13: F1(Z);
  14: F2(X,Z,Z);
  15: F3(X);
  16: F4(X,Y);
  17: G2(X,W,Z,
  18: W);
  19: 1 2 3 18 19
  ```

  then the output will be

  ```
  Error Code 5:  14 15 16 17 17 18
  ```

  Notice that line 17 is repeated in the output because both `X` and `W` on line 17 are not initialized before they appear in the polynomial evaluation `G2(X,W,Z,W)`. Also note that W appears twice as an argument and for each instance the corresponding line number is printed (17 and 18 respectively).

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

4 Program Output in the Absence of Errors

In the absence of errors, 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, I give an implementation guide that will help you plan your solution.

4.1 Variables and Locations

The program uses names to refer to variables. 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.

To support allocation of variables to `mem` entries, 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 to variable names). Initially the counter is 0. The first variable to be associated a location will get the location whose index is 0 (`mem`[0]) and the counter will be incremented to become 1. The next variable to be associated a location will get the location whose index is 1 and the counter will be incremented to become 2 and so on.

For example, if the input program is

```
1: POLY F1 =
2:            x^2 + 1;
3: POLY F2(x,y,z) = x^2 + y + z + 1;
4: POLY F3(y) = y^2 + 1;
5: POLY F4(x,y) = x^2 + y^2;
6: POLY G1 = x^2 + 1;
7: POLY F5 = x^2 + 1;
8: POLY G2(X,Y,Z,W) = X Y^2 + X Z + W + 1;
9: START
11: INPUT X;
12: INPUT Z;
13: F1(Z);
14: F2(X,Z,Z);
15: INPUT X;
16: INPUT Y;
17: INPUT Z;
18: F3(X);
19: F4(X,Y);
15: INPUT X;
16: INPUT Z;
17: INPUT W;
20: G2(X,Z,W
21: Z);
22: 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
```

4.2 Statements

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

4.2.1 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
```

4.2.2 Polynomial Evaluation

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

4.3 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.

5 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 test cases. Since this is the first project, the number of test cases provided with the project will be relatively large, but it will not be complete. It is still your responsibility to make sure that your implementation satisfies the requirements given in this document.

6 Instructions

Follow these steps:

- Make sure that you have read this document carefully.
- Make sure that you have read the *implementation guide* document carefully. It has detailed explanations about 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 that you have a clear overall picture of what the project will require before starting 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 (7.5.0) on **Ubuntu 18.04** (Ubuntu). **Your code must are compile with the GCC compiler and run on Ubuntu**, 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 **??** for more details.
- Submit your code through canvas.
- **Only the last version you submit is graded. There are no exceptions to this. You should not activate earlier version after submitting later versions. You should resubmit the earlier version not activate it! See detailed instruction in project 1 document.**

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 on gradescope, no other submission forms will be accepted.

7 Evaluation

The submissions are evaluated based on the automated test cases on the submission website. For the parsing test cases, as I explained in class, your submission should pass ALL the test cases to get credit for parsing. For the other categories, your grade will be proportional to the number of test cases that your submission passes.If your code does not compile when you submit it, 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. **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. **Error Code 1: 8%**. The grade for this category will be proportional to the test cases that are correctly handled by your program.
3. **Error Code 2: 8%**. The grade for this category will be proportional to the test cases that are correctly handled by your program.
4. **Error Code 3: 8%**. The grade for this category will be proportional to the test cases that are correctly handled by your program.
5. **Error Code 4: 8%**. The grade for this category will be proportional to the test cases that are correctly handled by your program.
6. **Error Code 5: 8%**. The grade for this category will be proportional to the test cases that are correctly handled by your program.
7. **No errors: 30%**.The grade for this category will be proportional to the test cases that are correctly handled by your program.