CSE340 Spring 2018 Project 1: Lexical Analysis

Due: **Friday, February 2, 2018 by 11:59 pm MST**

Introduction

The goal of this project is to show you how a description of a list of tokens can be used to automatically generate a lexical analyzer. The input to your program will be a description of tokens using regular expressions together with a text. Your program will generate data structures that will allow it to do lexical analysis on the text according to the tokens. The output of the program will be a list of tokens or an error message if the input does not have the correct format.

Input Format

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

```
input             →     tokens_section INPUT_TEXT
tokens_section    →     token_list HASH
token_list        →     token
token_list        →     token COMMA token_list
token             →     ID expr
expr              →     CHAR
expr              →     LPAREN expr RPAREN DOT LPAREN expr RPAREN
expr              →     LPAREN expr RPAREN OR LPAREN expr RPAREN
expr              →     LPAREN expr RPAREN STAR
expr              →     UNDERSCORE
```

Where

```
CHAR        =   a | b | ... | z | A | B | ... | Z  | 0 |  1 | ... | 9
LETTER      =   a | b | ... | z | A | B | ... | Z
SPACE       =  ' ' | \n | \t
INPUT_TEXT  =  " . (CHAR | SPACE)* . "
COMMA       = ','
LPAREN      = '('
RPAREN      = ')'
STAR        = '*'
DOT         =  '.'
OR          =  '|'
UNDERSCORE  =  '_'
ID          =  LETTER . CHAR*
```

In the description of regular expressions, UNDERSCORE represents epsilon.

Examples

The following are examples of input.

1. ```
   t1 (a)|(b) , t2 (a)* , t3 (((a)|(b))*).((c)*) #
   "a aa bb aab"
   ```

   This input specifies three tokens `t1`, `t2`, and `t3` and an INPUT_TEXT "a aa bb aab".

2. ```
   t1 (a)|(b) , t2 (a)* #
   "a aa bb aad aa"
   ```

   This input specifies two tokens `t1`, `t2`, and an INPUT_TEXT "a aa bb aad aa".

3. ```
   t1 (a)|(b) , t2 (a)* , t3 (((a)|(b))*).(((c)|(d))*)#
   "aaabbcaaaa"
   ```

   This input specifies three tokens whose names are `t1`, `t2`, and `t3` and an INPUT_TEXT "aaabbcaaaa".

Note

The input format can be confusing because there are two kinds of tokens.

1. There are tokens that are used in describing the input format such as ID and CHAR. When your program read them input, it will repeatedly call lexer.getToken() to get the sequence of tokens in the input and parse the various sections of the input according to the provided grammar.
2. The input itself has a description of tokens by providing the names of these tokens and the regular expressions of each token. These tokens whose description is provided in the input will be used to breakdown INPUT_TEXT (the last section of the input) as a sequence of tokens according to the `token_list` in the input description.

You should not confuse the two kinds of tokens.

Output Format

The output will be a sequence of tokens and their corresponding lexemes according to the list of tokens provided in the input or SYNTAX ERROR:

1. if the input to your program is in the correct format, the program should do lexical analysis on `INPUT_TEXT` and should produce a sequence of tokens and lexemes in `INPUT_TEXT` according to the list of tokens specified in the input to your program. Each token and lexeme should be printed on a separate line. The output on a given line will be of the form

   ```
   t ,  "lexeme"
   ```

   where t is the name of a token and lexeme is the actual lexeme for the token t. If during lexical analysis of `INPUT_TEXT`, a syntax error is encountered then ERROR is printed on a separate line and the program exits.

   In doing lexical analysis for `INPUT_TEXT`, `SPACE` is treated as a separator and is otherwise ignored.

2. if the input to your program is not in the correct format, the program should output SYNTAX ERROR and nothing else

Examples

Each of the following examples gives an input and the corresponding expected output.
   ```

1.
```
t1 (a)|(b) , t2 (a)* , t3 (((a)|(b))*).((c)*) #
"a aa bb aab"
```

   This input specifies three tokens `t1`, `t2`, and `t3` and an INPUT_TEXT "a aa bb aab". Since the input is in the correct format, the output of your program should be the list tokens in the INPUT_TEXT:

```
t1 , "a"
t2 , "aa"
t3 , "bb"
t3 , "aab"
```

2.
```
t1 (a)|(b) , t2 (a)* , t3 (((a)|(b))*).((c)*) #
"a aa bb aad aa"
```

   Since the input is in the correct format, the output of your program should be the list tokens in the INPUT_TEXT the output of the program should be

```
t1 , "a"
t2 , "aa"
t3 , "bb"
t2 , "aa"
ERROR
```

   Note that the input is in the correct format, but doing lexical analysis for INPUT_TEXT according to the list of tokens produces ERROR after the second `t2` token because there is no token that starts with `'d'`.

3.
```
t1a (a)|(b) , t2bc (a)* , t34 (((a)|(b))*).(((c)|(d))*)#
"aaabbcaaaa"
```

   This input specifies three tokens whose names are `t1a`, `t2bc`, and `t34` and an input text "aaabbcaaaa". Since the input is in the correct format, the output of your program should be the list tokens in the INPUT_TEXT:

```
t34 , "aaabbc"
t2bc , "aaaa"
```

Requirements

You should write a program to generate the correct output for a given input as described above.

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.

In your solution, you are not allowed to use any built-in or library support for regular expressions in C/C++. This requirement will be enforced by checking your code.

How to Implement a Solution

The main difficulty in coming up with a solution is to transform a given list of token names and their regular expression descriptions into a `getToken()` function for the given list of tokens. This transformation will be done in three high-level steps:

1. Transform a regular expression into an NFA. The goal here is to parse a regular expression description and generate a graph that represents the regular expression[1]. The generated graph will have a specific format and I will describe below how to generate it. In what follows I will call it a *regular expression graph*, or REG for short.

[1] The graph is a representation of a non-deterministic finite state automaton

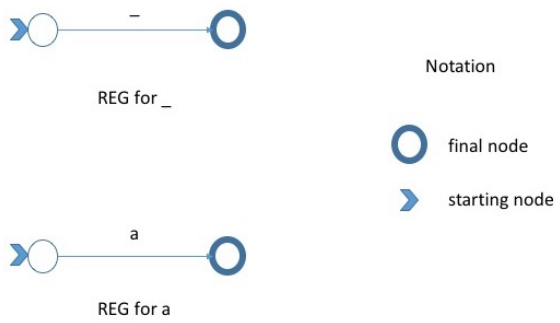

Figure 1: Regular expressions graphs for the base cases

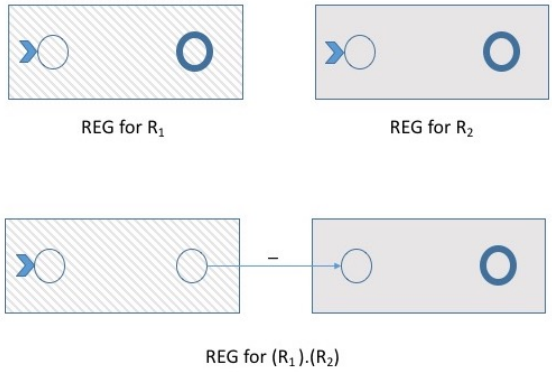

Figure 2: Regular expression graph for the an expression obtained using the dot operator

2. Write a function `match(r,s,p)`, where `r` is a REG , `s` is a string and `p` is a position in the string `s`. The function match will return the longest possible lexeme starting from position `p` in the string `s` that matches the regular expression of the graph `r`.
3. Write a class `LexicalAnalyzer(list,s)` where `list` is a list of structures: {`token_name`, *reg_pointer*} and `s` is an input string. `LexicalAnalyzer` stores the list of structures and keeps track of the part of the input string that has been processed. Lexical analyzer has a method `getToken()`. For every call of `getToken()`, `match(r,s,p)` is called for every REG r in the list starting from the current position `p` maintained in the lexical analyzer. `getToken()` returns the token with the longest matching prefix together with its lexeme longest and updates the current position. If the longest matching prefix matches more than one token, the matched token that is listed first in the list of tokens is returned.

In what follows I describe how a regular expression description can be transformed into a REG and how to implement the function `match(r,s,p)`.

Constructing REGs

The construction of REGs is done recursively. The construction we use is called Thompson's construction. Each REG has a one starting node and one final node. For the base cases of epsilon and `a`, the REGs are shown in Figure 1. For the recursive cases, the constructions are shown in Figures 2, 3, and 4. An example REG for the regular expression `((a)*).((b)*)` is shown in Figure 5.

Data Structures and Code for REGs

In the construction of REGs, every node has at most two outgoing arrows. This will allow us to use a simple representation of a REG node.

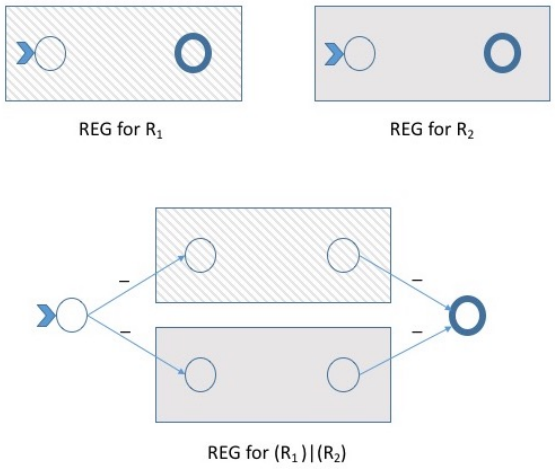

Figure 3: Regular expression graph for the an expression obtained using the or operator

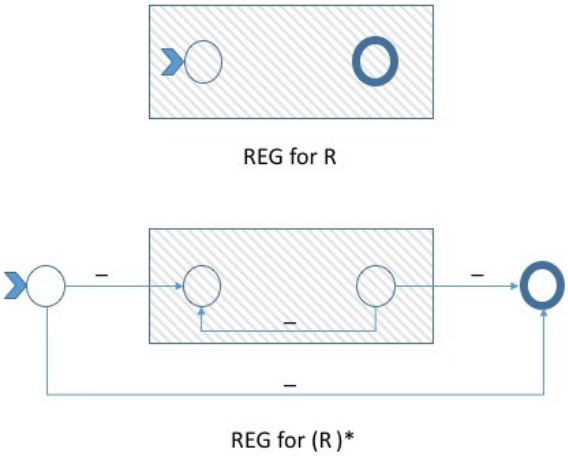

Figure 4: Regular expression graph for the an expression obtained using the star operator

```
struct REG_node {
 struct REG_node * first-neighbor;
 char first_label;
 struct REG_node * second_neighbor;
 char second_label;
}
```

In the representation, `first_neighbor` is the first node pointed to by a node and `second_neighbor` is the second node pointed to by a node. `first_label` and `second_label` are the labels of the arrows from the node to its neighbors. If a node has only one neighbor, then `second_neighbor` will be NULL. If a node has no neighbors, the both `first_neighbor` and `second_neighbor` will be NULL.

```
struct REG {
   struct REG_node * starting;
   struct REG_node * accepting;     // note that I changed final to accepting because final
                                    // is reserved in C++
}
```

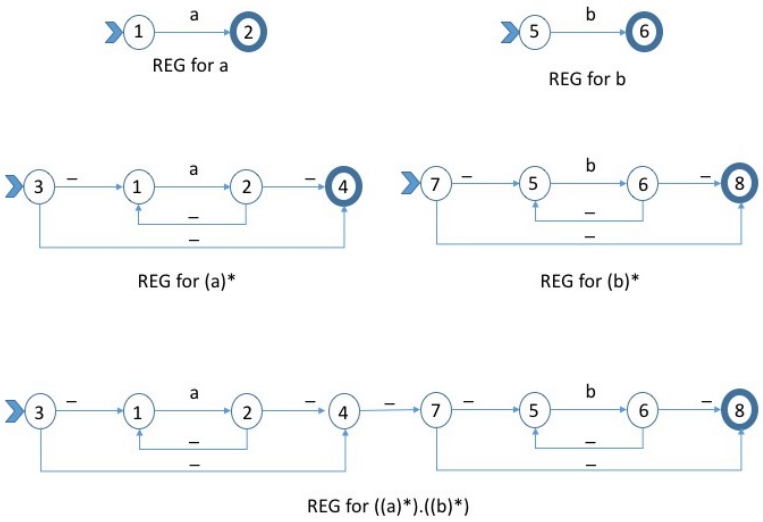

Figure 5: Regular expression graph for the an expression obtained using concatenation and star operators

The function `parse_regular_expr()` should not only parse a regular expression, but should also return the REG of the regular expression that is parsed. The construction of REGs is done recursively. An outline of the process is shown on the next page.

```
struct REG * parse_regular_expression()
{
   // if expression is UNDERSCORE or a CHAR, say 'a' for example
   // create a REG for the expression and return it
   // (see Figure 1, for how the REG looks like)

   // if expression is (R1).(R2)
   //    the program will call parse_regular_expr() twice, once
   //    to parse R1 and once to parse R2
   //
   //    Each of the two calls will return a REG, say they are
   //    r1 and r2
   //
   //    construct a new REG r for (R1).(R2) using the
   //    two REGs r1 and r2
   //    (see Figure 2 for how the two REGs are combined)
   //
   //    return r
   //
   //    the cases for (R1)|(R2) and (R)* are similar
}
```

Detailed Examples for REG Construction

I consider the regular expression `((a)*).((b)*)` and explain step by step how its REG is constructed (Figure 5).

When parsing the `((a)*).((b)*)`, the first expression to be parsed is `a` and its REG is constructed according to Figure 1. In Figure 5, the nodes for the REG of the regular expression `a` have numbers 1 and 2 to indicate that they are the first two nodes to be created.

The second expression to be parsed when parsing `((a)*).((b)*)` is `(a)*`. The REG for `(a)*` is obtained from the REG for the regular expression `a` by adding two more nodes (3 and 4) and adding the appropriate arrows as described in the general case in Figure 4. The starting node for the REG of `(a)*` is the newly created node 3 and the accepting node is the newly created node 4.

The third regular expression to be parsed while parsing `((a)*).((b)*)` is the regular expression `b`. The REG for regular expression `b` is constructed as shown in Figure 1. The nodes for this REG are numbered 5 and 6.

The fourth regular expression to be parsed while parsing `((a)*).((b)*)` is `(b)*`. The REG for `(b)*` is obtained from the REG for the regular expression `b` by adding two more nodes (7 and 8) and adding the appropriate arrows as described in the general case in Figure 4. The starting node for the REG of `(b)*` is the newly created node 7 and the accepting node is the newly created node 8.

Finally, the last regular expression to be parsed is the regular expression `((a)*).((b)*)`. The REG of `((a)*).((b)*)` is obtained from the REGs of `(a)*` and `(b)*` by creating a new REG whose initial node is node 3 and whose accepting node is node 8 and adding an arrow from node 4 (the accepting node of the REG of `(a)*`) to node 7 (the initial node for the REG of `(b)*`).

Another example for the REG of `(((a)*).((b).(b)))|((a)*)` is shown in Figures 7 and 8. In the next section, I will use this example to illustrate how `match(r,s,p)` can be implemented.

Implementing `match(r,s,p)`

Given an REG `r`, a string `s` and a position `p` in the string `s`, we would like to determine the longest possible lexeme that matches the regular expression for `r`.

As you will see in CSE355, a string `w` is in L(`R`) for a regular expression `R` with REG `r` if and only if there is a path from the starting node of `r` to the accepting node of `r` such that `w` is equal to the concatenation of all labels of the edges along the path. I will not into the details of the equivalence in this document. I will describe how to find the longest possible substring `w` of `s` starting at position `p` such that there is a path from the starting node of `r` to the accepting node of `r` that can be labeled with `w`.

To implement `match(r,s,p)`, we need to be able to determine for a given input character `a` and a set of nodes `S` the set of nodes that can be reached from nodes in `S` by *consuming* `a`. To consume `a` we can traverse any number of edges labeled '_', traverse one edge labeled `a`, then traverse any number of edges labeled '_'.

At each step, the solution will keep track of the set of nodes `S` that can be reached by consuming a prefix of the input string so far. Initially `S` is the set of nodes that can be reached from the starting node by not consuming any input (empty prefix).

For a given character `a` and a given set `S`, the solution will find all the nodes that can be reached from `S` by consuming `a`. This will be implemented in a function that can be called `match_one_char()` shown in Figure 6.

To implement `match(r,s,p)`, we start with the set of nodes that can be reached from the starting node of `r` by consuming no input. Then we repeatedly call `match_one_char()` for successive characters of the string `s` starting from position `p` until the returned set of nodes `S` is empty or we run out of input. If at any point during the repeated calls to `match_one_char()` the set `S` of nodes contains the accepting node, we note the fact that the prefix of string `s` starting from position `p` up to the current position is matching. At the end of the calls to `match_one_char()` when `S` is empty or the end of input is reached, the last matched prefix is the one returned by `match(r,s,p)`. If none of the prefixes are matched, then there is no match for `r` in `s` starting at `p`.

```
set_of_nodes match_one_char(set_of_nodes S, char c)
{

      // 1. find all nodes that can be reached from S by consuming c
      //
      //    S' = empty set
      //    for every node n in S
      //        if  ( (there is an edge from n to m labeled with c) &&
      //              ( m is not in S) ) {
      //             add m to S'
      //        }
      //
      //     if (S' is empty)
      //         return empty set
      //
      //    At this point, S' is not empty and it contains the nodes that
      //    can be reached from S by consuming the character c directly
      //
      //
      // 2. find all all nodes that can be reached from the resulting
      //    set S' by consuming no input
      //
      //     changed = true
      //     while (changed)  {
      //        changed = false
      //        for every node n in S'
      //             if  ( (there is an edge from n to m labeled with '_') &&
      //                   ( m is not in S') ) {
      //                add m to S'
      //                changed = true
      //             }
      //     }
      //
      //     at this point the set S' contains all nodes that can be reached
      //     from S by first consuming C, then traversing 0 or more epsilon
      //     edges
      //
      //     return S'
 }
```

Figure 6: Pseudocode for matching one character

.

Note. The algorithms given above are not the most efficient, but they are probably the simplest to implement the matching functions.

Detailed Example for Implementing `match(r,s,p)`

In this section, I illustrate the steps of an execution of `match(r,s,p)` on the REG of `(((a)*).((b).(b)))|((a)*)` shown in Figure 8. The input string we will consider is the string `s = "aaba"` and the initial position is `p = 0`.

1. Initially
   The set of states that can reached by consuming no input starting from node 17 is $S_0 = \{17, 3, 1, 4, 9, 15, 13, 16, \mathbf{18}\}$

   Note that S_0 contains node **18** which means that the empty string is a matching prefix.

2. Consuming `a`
   The set of states that can be reached by consuming `a` starting from S_0 is $S_1 = \{2, 14\}$

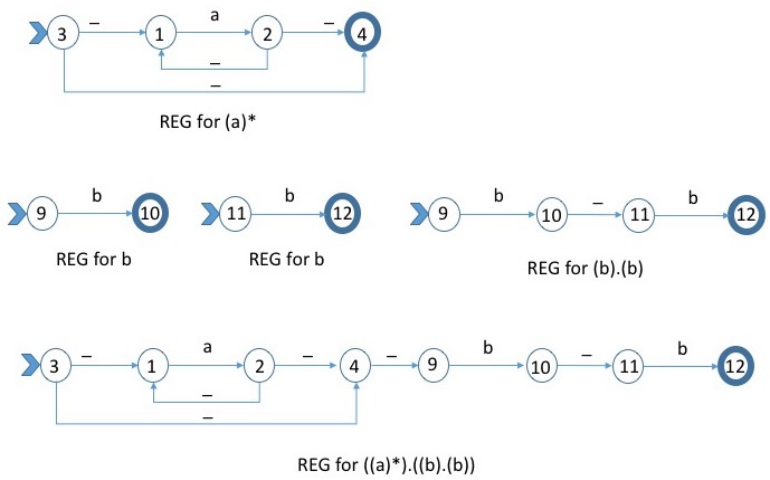

Figure 7: Regular expression graph `((a)*).((b).(b))`

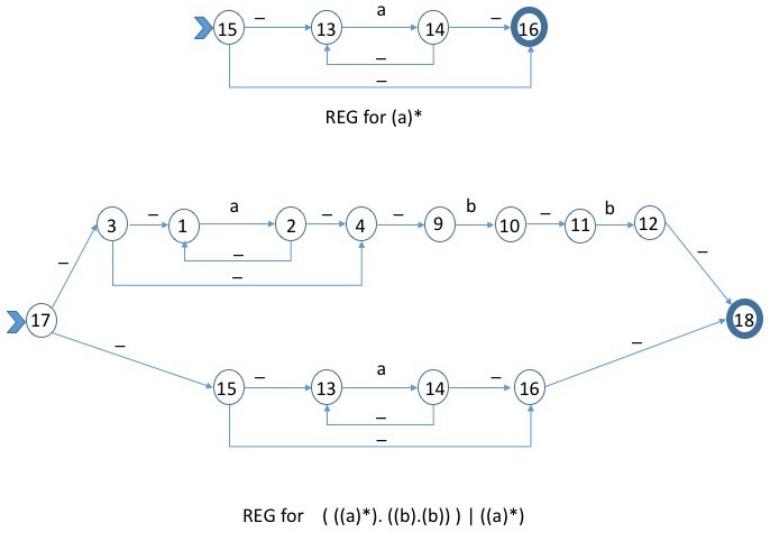

Figure 8: Regular expression graph `(((a)*).((b).(b)))|((a)*)`

The set of states that can be reached by consuming no input starting from S_1 is $S_{1\_} = \{2, 1, 4, 9, 14, 13, 16, \mathbf{18}\}$

Note that $S_{1\_}$ contains node **18**, which means that the prefix `"a"` is a matching prefix.

3. Consuming `a`
   The set of states that can be reached by consuming `a` starting from $S_{1\_}$ is $S_2 = \{2, 14\}$
   The set of states that can be reached by consuming no input starting from S_2 is $S_{2\_} = \{2, 1, 4, 9, 14, 13, 16, \mathbf{18}\}$

   Note that $S_{2\_}$ contains node **18**, which means that the prefix `"aa"` is a matching prefix.

4. Consuming `b`
   The set of states that can be reached by consuming `b` starting from $S_{2\_}$ is $S_3 = \{10\}$
   The set of states that can be reached by consuming no input starting from S_3 is $S_{3\_} = \{10, 11\}$

   Note that $S_{3\_}$ does not contain node **18** which means that `"aab"` is not a matching prefix, but is still a viable prefix.

5. Consuming `a`

The set of states that can be reached by consuming `a` starting from $S_{3\_}$ is $S_4 = \{\}$

Since S_4 is empty, `"aaba"` is not viable and we stop.

The longest matching prefix is `aa`. This is the lexeme that is returned. Note that the second call to `match(r,s,p)` starting after `"aa"` will return ERROR.

Instructions

Follow these steps:

- Download the `lexer.cc`, `lexer.h`, `inputbuf.cc` and `inputbuf.h` files accompanying this project description. Note that these files might be a little different from the code you've seen in class or elsewhere.
- Compile your code using GCC version 4.8.5 compiler on `CentOS 7`. You will need to use the `g++` command to compile your code in a terminal window. See section 4 for more details on how to compile using GCC.

Note that you are required to compile and test your code on CentOS 7 using the GCC compiler version 4.8.5. You are free to use any IDE or text editor on any platform, however, using tools available in CentOS (or tools that you could install on CentOS) could save time in the development/compile/test cycle.

- 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 test script `test1.sh`. See section 5 for more details.
- Submit your code on the course submission website before the deadline. You can submit as many times as you need. Make sure your code is compiled correctly on the website, if you get a compiler error, fix the problem and submit again.
- **Only the last version you submit is graded. There are no exception to this.**

Keep in mind that

- You should use C/C++, no other programming languages 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 the course submission website, no other submission forms will be accepted.
- You should familiarize yourself with the CentOS environment and the GCC compiler. Programming assignments in this course might be very different from what you are used to in other classes.

3. Evaluation

The submissions are evaluated based on the automated test cases on the submission website. Your grade will be proportional to the number of test cases passing. If your code does not compile on the submission website, you will not receive any points.

NOTE: The next two sections apply 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.

4. Compiling your code with GCC

You should compile your programs with the GCC compilers which are available in CentOS 7. GCC is a collection of compilers for many programming languages. There are separate commands for compiling C and C++ programs:

- Use the `gcc` command to compile 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 folder as the program. You can change the output file name by specifying the `-o` option:

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

To enable C++11 with `g++`, use the `-std=c++11` option:

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

The following table summarizes some useful GCC compiler options:

Switch	Can be used with	Description
`-o path`	`gcc`, `g++`	Change the filename of the generated artifact
`-g`	`gcc`, `g++`	Generate debugging information
`-ggdb`	`gcc`, `g++`	Generate debugging information for use by GDB
`-Wall`	`gcc`, `g++`	Enable most warning messages
`-w`	`gcc`, `g++`	Inhibit all warning messages
`-std=c++11`	`g++`	Compile C++ code using 2011 C++ standard
`-std=c99`	`gcc`	Compile C code using ISO C99 standard
`-std=c11`	`gcc`	Compile C code using ISO C11 standard

You can find a comprehensive list of GCC options in the following page:

https://gcc.gnu.org/onlinedocs/gcc-4.8.5/gcc/

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 and the final executable will be `a.out`.

You can replace `g++` with `gcc` in all examples listed above to compile C programs.

5. Testing your code on CentOS

Your programs should not explicitly open any file. You can only use the **standard input** e.g. `std::cin` in C++, `getchar()`, `scanf()` in C and **standard output** e.g. `std::cout` in C++, `putchar()`, `printf()` in

C for input/output.

However, this restriction does not limit our ability to feed input to the program from files nor does it mean that we cannot save the output of the program in a file. We use a technique called standard IO redirection to achieve this.

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 feed 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. 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. Note that programs have access to another standard stream which is called standard error e.g. `std::cerr` in C++, `fprintf(stderr, ...)` 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.

Finally, it's possible to mix both into one command:

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

Which will redirect standard input and standard output to `input_data.txt` and `output_file.txt` respectively.

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

Test Cases

A test case is an input and output specification. For a given input there is an *expected* output. A test case for our purposes is usually 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
```

The output generated by the program will be stored in `program_output.txt`. To see if the program generated the expected output, we need to compare `program_output.txt` and `test_name.txt.expected`. We do that using a general purpose tool called `diff`:

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

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

Our grading system uses this method to test your submissions against multiple test cases. There is also a test script accompanying this project `test1.sh` which will make your life easier by testing 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 folder as your project source files
- Open a terminal window and navigate to your project folder
- Unzip the test archive using the `unzip` command: `bash  $ unzip test_cases.zip`

NOTE: the actual file name is probably different, you should replace `test_cases.zip` with the correct file name.

- 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`).