CSE340 Spring 2019 Project 1: A Simple Compiler!

Due: **Friday, February 1, 2019 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 go into a detailed description on how to go about achieving these tasks in subsequent sections. The goal of this project is to implement the function a simple compiler for a simple programming language. By implementing this simple compiler, you will do some basic parsing and use some basic 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 procedure declarations followed by a main procedure.
2. The second part of the input is a sequence of integers which will be used as the input to the program.

Your compiler will read the program, represent it internally in appropriate data structures, and then executes the program using the second part of the input as the input to the program. The output of the compiler is the output produced by the program execution. More details about the input format and the expected output of your program are given in subsequent sections.

The remainder of this document is organized as follows.

1. The second section describes the input format.
2. The third section describes the expected output.
3. The fourth section describes the requirements on your solution.
4. The fifth section gives instructions for this programming assignment and additional instructions that apply to all programming assignments in this course.

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              →    main
program              →    proc_decl_section main
proc_decl_section    →    proc_decl proc_decl_section
proc_decl            →    PROC procedure_name procedure_body ENDPROC
procedure_name       →    ID
procedure_name       →    NUM
procedure_body       →    statement_list
statement_list       →    statement
statement_list       →    statement statement_list
statement            →    input_statement
statement            →    output_statement
statement            →    procedure_invocation
```

```
statement               →    do_statement
statement               →    assign_statement
input_statement         →    INPUT ID expr SEMICOLON
output_statement        →    OUTPUT ID expr SEMICOLON
procedure_invocation    →    ID SEMICOLON
do_statement            →    DO ID procedure_invocation
assign_statement        →    ID EQUAL expr SEMICOLON
expr                    →    primary
expr                    →    primary operator primary
operator                →    PLUS
operator                →    MINUS
operator                →    MULT
operator                →    PLUS
primary                 →    ID
primary                 →    NUM
main                    →    MAIN procedure_body
inputs                  →    NUM
inputs                  →    NUM inputs
```

The code that we provided has a class `LexicalAnalyzer` with methods `getToken()` and `ungetToken()`. You do not need to change the function. You parser will use the provided functions to get or unget tokens as needed. The definition of the tokens is given below for completeness. To use the methods, you should first instantiate a (`lexer` object of class `LexicalAnalyzer` and call the methods on this instance.

```
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      =  ;
PLUS           =  +
MINUS          =  -
MULT           =  *
DIV            =  /
MAIN           =  (M).(A).(I).(N)
PROC           =  (P).(R).(O).(C)
ENDPROC        =  (E).(N).(D).(P).(R).(O).(C)
INPUT          =  (I).(N).(P).(U).(T)
OUTPUT         =  (O).(U).(T).(P).(U).(T)
DO             =  (D).(O)
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 `program` on the `inputs`. For now do not worry how that is achieved. I will explain that in details.

2.2 Examples

The following are examples of input with corresponding outputs. The output will be explained in more details in the next section.

1.

```
MAIN
X = 1;
Y = X;
OUTPUT X;
OUTPUT Y;
1 2 3 18 19
```

This example shows a program with no procedure declarations (PROC) and a MAIN procedure that does not read any input. The output of the program will be

```
1 1
```

The sequence of numbers at the end does not affect the output of the program.

2.

```
MAIN
INPUT X;
INPUT Y;
X = X + Y;
Y = X+Y;
OUTPUT X;
OUTPUT Y;
3 7 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 3. The second `input` statement reads a value for `Y` which gets the value 7. Here the output will be

```
10 17
```

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

3.

```
PROC INCX
X = X + 1;
ENDPROC
MAIN
INPUT X;
INPUT Y;
INCX;
INCX;
Y = X+Y;
OUTPUT Y;
OUTPUT X;
3 18 19
```

In this example, we have a procedure called `INCX`. When the procedure is invoked, the code of the procedure is executed. In this case, `X` is incremented by 1. The second invocation also increments `X` again so the final value of `X` is 5. The output is the following.

```
23 5
```

4.

```
PROC INCX
X = X + 1;
ENDPROC
MAIN
INPUT X;
INPUT Y;
Z = 2;
DO Z INCX;
Y = X+Y;
OUTPUT Y;
OUTPUT X;
3 18 19
```

This is similar to the previous example, but instead of invoking `INCX` two separate times, we achieve the same result with a `do_statement`. `Z = 2` assigns the value 2 to `Z` and `DO Z INCX;` will invoke `INCX` Z times (with the value of `Z` equal to 2).

For your parser, the parse function do not necessarily return true or false. They can return other quantities. For example it might be useful for `parse_primary()` to return the token of the primary.

3 Semantics

In this section I give a precise definition of the meaning of the input and the output that your compiler should generate.

3.1 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. All variables are initially 0. 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 simple location table (or map) that associates a variable name with a location. As you parse the program, if you encounter a new variable name, you give it a new location and add an entry to in the location table with the variable name and location. In order to keep track of available locations, you can use a global variable `location` which is initially 0 and which is incremented every time a variable is allocated a location.

For example, if the input program is

```
MAIN
INPUT X;
INPUT Y;
X = X + Y;
Y = X+Y;
Z = X+Y;
W = Z;
OUTPUT X;
OUTPUT Y;
3 7 18 19
```

Then the locations of variables will be

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

3.2 Statements

We explain the semantics of each statement in terms of an implementation model that assigns locations to variables and we have described in the previous section.

3.2.1 Assignment Statement

We consider an assignment of the form

```
ID = primary1 operator primary2
```

This has the following effect

```
mem[location(t1.lexeme)] = value(primary1) operator value(primary2)
```

where `t1` is the ID token for the lhs for the assignment and value(primary) is equal to `atoi(primary.lexeme)` or `mem[location(primary.lexeme)]` depending on whether or not `primary` is NUM or ID respectively.

For example, if the input program is

```
MAIN
INPUT X;
INPUT Y;
X = Y + 3;
Y = X+Y;
Z = X+Y;
W = Z;
OUTPUT X;
OUTPUT Y;
3 7 18 19
```

the statement `X = Y+3` will be equivalent to `mem[0] = mem[1] + 3`.

3.2.2 Input and Output

Input and output statements are relatively straightforward

- `OUTPUT X` is equivalent to `cout « mem[location("X")];`
- `INPUT X` is equivalent to `cin » mem[location("X")];`.

3.2.3 Procedure Invocation

A procedure invocation

```
ID ;
```

is equivalent to executing every statement in the procedure at the point of the call. The execution of

```
stmt1
stmt2
...
stmtk
P;
stmtk+1
stmtk+2
stmtm
```

where `P` is the name of a procedure declared as

```
PROC P
stmt'1
stmt'2
...
stmt'm'
ENDPROC
```

is equivalent to

```
stmt1
stmt2
...
stmtk
stmt'1
stmt'2
...
stmt'm'
stmtk+1
stmtk+2
stmtm
```

3.2.4 DO Statememt

The statement `DO ID1 ID2;` where the first identifier is the name of a variable and the second identifier is the name of a procedure is equivalent to n invocations of `ID2` where n is the value of `ID1` at the point the `do_statement` is executed. If the variable that determine the number of invocations is modified in the procedure body, that does not affect the number of invocations. For example

```
PROC P
X = X+1;
ENDPROC
MAIN
INPUT X;
DO X P;
OUTPUT X;
3
```

The statement `DO X P;` is equivalent to `P; P; P;`. The output of the program will be 6.

3.3 Assumptions

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

1. Two procedures declared with the same name. You can assume that all procedure names are unique. So, an invocation of a procedure cannot be ambiguous.
2. You can assume that if there is an invocation with a given procedure name, there must be a procedure declaration with the same name.
3. You can assume that if there is an invocation with a given procedure name, then there is no variable with the same name in the program.
4. You can assume that if there is a variable with a given name in the program, then there is no procedure declaration for a procedure with the same name as the variable.
5. 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 the case when `location` variable reaches 1000 because that is good programming practice.

In addition, you can assume the following that the input will not have recursive procedure invocations.

4 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.

5 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 on **Ubuntu 18** (Ubuntu). 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 Ubuntu using the GCC compiler.You are free to use any IDE or text editor on any platform, however, using tools available on Ubuntu (or tools that you could install on Ubuntu) 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 Ubuntu environment and the GCC compiler. Programming assignments in this course might be very different from what you are used to in other classes.

5.0.1 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.

5.0.2 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 `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.

5.0.3 Testing your code on Ubuntu

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`).