CS4100/5100 COMPILER PROJECT
Part 1: Foundations

The purpose of this assignment is to give the student a practical introduction to the structure and usage of the foundational elements of the compiler project. The code developed here will be used throughout the rest of the semester. **Creating the ADT's given below, exactly as described, is crucial to the success of the project!** While there is room for individuality in the actual *implementation*, it is necessary that each structure utilize exactly the ***interfaces*** and ***storage capabilities*** described, in order for them to be used effectively and efficiently in the development of the rest of the project. **Use the same language which you will use to implement the rest of the compiler project: C++, C#, or Java. Read all requirements below carefully!**

Program Objectives: Implement a program which will 1) Create a Quad Table instance and fill it with the Quad codes developed to calculate 10-summation and 10-factorial (discussed in class); 2) Create a ReserveTable and store the opcode mnemonics in it; 3) Create and populate a Symbol Table to accommodate the data elements needed to execute the Quad codes correctly; 4) Run the Quad codes using the interpreter algorithm given, and print the final answers for the problem.

Main Program:
The main program for this part of the project should be a simple, modular sequence of calls to other methods/function, essentially accomplishing the following:

```
BuildQuads;          //Allocate and initialize contents of
                     //Quad table, including the associated
                     //mnemonics in a ReserveTable & PRINT
                     //them both
BuildSymbolTable;    //Allocate and initialize the Symbol
                     //table and PRINT it
                     //Run the interpreter in TRACE mode
InterpretQuads(Quadtable,SymbolTable,True);
                     //Run the interpreter without TRACE
InterpretQuads(Quadtable,SymbolTable,True);
```

ADTs To Implement:
The following ADTs must be established according to the requirements below:

1) SymbolTable
The symbol table conceptually consists of an indexed list of entries, with each entry containing a *name* field (string type); a *kind* field (to hold a label, variable, or constant indicator); a *data_type* field specifying the data as an integer, float, or string; and a set of *3 value* fields to hold an integer, float, and string (labels will store an integer quad address, variables and constants will have a float, integer, or string storage area). The interface to the SymbolTable must implement the following methods (in addition to a constructor, as needed)**:**

int AddSymbol(String symbol, int kind, int value)
int AddSymbol(String symbol, int kind, double value)
int AddSymbol(String symbol, int kind, String value)

Adds *symbol* with given *kind* and *value* to the symbol table, automatically setting the correct *data_type,* and returns the index where the symbol was located. **If the symbol is already in the table, no change or verification is made, and this**

just returns the index where the symbol was found.

int LookupSymbol(String symbol)
Returns the index where symbol is found, or -1s if not in the table

symboldata GetSymbol(int index)
Return kind, data type, and value fields stored at index

UpdateSymbol(int index, int kind, int value)
UpdateSymbol(int index, int kind, double value)
UpdateSymbol(int index, int kind, String value)
Set appropriate fields at slot indicated by index

PrintSymbolTable()
Prints the utilized rows of the symbol table in neat tabular format, showing only the value field which is active for that row

2) QuadTable
The QuadTable is different from the SymbolTable in its access and contents. Each indexed entry row consits of four int values representing an opcode and three operands. The methods needed are:

Initialize(size and other parameters as needed)
Create a new, empty QuadTable ready for data to be added, with the specified number of rows (*size*).

int NextQuad()
Returns the int index of the **next open slot in the QuadTable.** Very important during code generation, this must be implemented exactly as described.

void AddQuad(int opcode, op1, op2, op3)
Expands the active length of the quad table by adding a new row at the *NextQuad* slot, with the parameters sent as the new contents, **and increments the NextQuad counter to the next available (empty) index.**

quadstruct GetQuad(int index)
Returns the data for the opcode and three operands located at *index*

void SetQuad(int index, opcode, op1, op2, op3)
Changes the contents of the existing quad at *index.* Used only when backfilling jump addresses later, during code generation, and very important

String GetMnemonic(int opcode)
Returns the mnemonic string (‘ADD’, ‘PRINT’, etc.) associated with the *opcode* parameter. Used during interpreter ‘TRACE’ mode to print out the stored opcodes in readable format. Use the ReserveTable ADT to implement this.

PrintQuadTable()
Prints the currently used contents of the Quad table in neat tabular format

3) ReserveTable
This is a lookup table ADT that will be used here for the opcode lookup, and later in the compiler, a separate instance will hold the reserved word list for the language. Make sure it is built so that it can be used for both applications. Each indexed entry is a pair consisting of a name string and an integer code. The table as we use it is static, and intialized once at the start of the program, and then used only for lookups later on. The needed methods are:

Initialize()
Constructor, as needed.

int Add(string name, int code)
Returns the index of the row where the data was placed; just adds to end of list.

int LookupName(String name)
Returns the *code* associated with *name* if *name* is in the table, else returns -1

string LookupCode(int code)
Returns the associated *name* if *code* is there, else an empty string

PrintReserveTable()
Prints the currently used contents of the Reserve table in neat tabular format

4) The Interpreter
The interpreter should be called like this:

InterpretQuads(Quadtable Q, SymbolTable S, boolean TraceOn);

so that the Quad table and Symbol table are inputs to the interpreter. Your program must implement the full 'Interpreter' functionality from the ***Intro to Quad Codes*** handout. Translate it from the pseudo-code handed out into the language to be used for your compiler project. The interpreter function must have access to both your symbol table structure and your quad table structure. The interpreter must be able tp produce an echo 'trace mode' printout, which simply indicates the current PC (program counter) and quad data that will be executed next, like:

```
PC = 0025: ADD 5, 8, 2
PC = 0026: BNZ 12
```

Note that this will be used for debugging the compiler-generated code at the end of the entire project.

Structure of the code
In order to receive full points for this program, it must utilize good Software Engineering practices, have full documentation, and be logically structured. The code must show good encapsulation and independence of the various structures, and conform to the interfaces described above. Meaningful variable names and other good practices, such as clear comments within the code, are expected to be at a professional level.

Turn-In Requirements
As a **single text file,** collect and **on Canvas,** turn in all source code, neatly documented and logically arranged. Neatness counts! High quality coding standards are implicit in a senior/graduate level course. Also include, in the same plain .txt format file, the run of the program, illustrating clearly in the console output from the execution of the interpreter in both its trace and non-trace modes. **See the Grading Sheet for exact turn-in requirements and point values for each graded item.**