**CS323 Documentation**

**Problem Statement**

There were two parts to assignment three. Part one asks to construct a symbol table to handle the defined symbols as well as perform symbol lookups. Each entry in the table holds a lexeme and a corresponding pseudo ‘memory address’ where an identifier is placed within the symbol table. There are procedures to see if a particular identifier is already in the table, a procedure that inserts a symbol into the table, and a procedure that prints out the contents of all identifiers declared in the table. If an identifier is used without declaration, then the parser provides a meaningful message and exits the program. Also, if an identifier is already in the table and wants to declare it for the second time, the parser provides an error message. The second part of assignment three asks to generate assembly code. Adding instructions to the syntax analyzer from assignment 2 will do this. Instructions can be found on the description for assignment 3. Also, partial solutions were provided by Dr. Choi to help guide the completion of our top down parser.

**How to use your program**

The program has only been tested on the MacOSX operating system running the newest version of Python3.4, any incompatibility will be due to the user not using the correct version of python. The program requires python version greater than 2.5 due to the use of ternary operators.

The data.txt file, which contains the input script, must be in the same directory as the OC.py python script. To run the program, enter the directory of the data file and python script and type: $ python3.4 OC.py

Once the program is opened, it will prompt the user for input:

‘Enter file you would like to open (type “quit” to exit): ’

You may enter the name of a file you would like to input such as: testcase1.txt

Once user inputs file, the program executes. Once the object code generator is finished, the program prompts the user whether they would like to process another file. ‘yes’ input will ask user for another input file, ‘no’ input will exit the program.

Once program quits, the contents of the lexer, syntax analyzer, and object code generator are saved to the working directory as ‘data.RAT’, ‘data.SA’, and ‘data.OC’ respectively.

If a file is not recognized the program will respond with:

‘Your file was not found!’

Then it will prompt user ‘Would you like to process another file? (yes/no):’

‘no’ entry will exit the program. ‘yes’ will ask user for another data file input.

To quit the program type: quit into the terminal and the program will exit.

Successful runs will output to the terminal and be written into a file with the name of the input (user defined) file and the extension “.RAT”, “.SA”, and “.OC”.

Once the program is called, it will run on a loop until the user enters: quit

If the syntax analysis has an error, the error is reported to the terminal as well as stored in a file.

Example run:

Arts-MBP:ASGN3\_OC Arty$ python3.4 OC.py

Enter file you would like to open (type "quit" to exit): testcase1.txt

Lexer working...

File open!

...Lexer complete!

Your Tokens and Lexemes have been saved as testcase1.RAT in the working directory.

Object Code Generator running...

...Object Code Generator finished!

There were no errors!

Your syntactic analysis of testcase1 has been saved as testcase1.SA in the working directory.

Your object code of testcase1 has been saved as testcase1.OC in the working directory.

Would you like to process another file? (yes/no): yes

Enter file you would like to open (type "quit" to exit): testcase2.txt

Lexer working...

File open!

...Lexer complete!

Your Tokens and Lexemes have been saved as testcase2.RAT in the working directory.

Object Code Generator running...

...Object Code Generator finished!

There were no errors!

Your syntactic analysis of testcase2 has been saved as testcase2.SA in the working directory.

Your object code of testcase2 has been saved as testcase2.OC in the working directory.

Would you like to process another file? (yes/no): yes

Enter file you would like to open (type "quit" to exit): testcase3.txt

Lexer working...

File open!

...Lexer complete!

Your Tokens and Lexemes have been saved as testcase3.RAT in the working directory.

Object Code Generator running...

...Object Code Generator finished!

There were no errors!

Your syntactic analysis of testcase3 has been saved as testcase3.SA in the working directory.

Your object code of testcase3 has been saved as testcase3.OC in the working directory.

Would you like to process another file? (yes/no): no

Goodbye!

**Design of your program**

For this assignment, code from assignment 2 (syntax analyzer) was modified to fit the specifications of assignment 3. There were 4 class objects that needed to be added: Instruction, Instr\_table, Symbol, Symbol\_Table. These objects helped generate, manage, and store the object code.

The ‘Symbol’ class was designed to store an individual symbol in the symbol table. This class stores and modifies the global variable ‘\_MEMORY’, stores the type of the symbol, and the lexeme associated by the symbol.

The ‘Symbol\_Table’ class holds the instances of symbols. This table is initialized as an empty list to which symbols are appended. There are four functions which manage the symbol table: insert, look\_up\_lex, verify, list. The insert method checks to see if the symbol already exists in the symbol table by using the look\_up\_lex method and then creates and appends the instance of a new symbol. If a symbol already exists in the symbol table and the insert function is called, then a meaningful error is printed to the user and the program exits. The look\_up\_lex function checks every item in the symbol table against a given lexeme. If found, the item’s address is returned, if not found, a Boolean ‘False’ is returned. The verify function uses the look\_up\_lex function to verify if the variable is in the table. If the variable is found then the address is returned, if not found then a meaningful error is generated and the program aborts. The list function formats and prints every symbol in the symbol table with the following information: lexeme, address, type.

The ‘Instruction’ object holds information about a single instruction instance. The private variables in the objects include: address, operation, and operand. There are 3 pairs of get/set methods for the three private variables. The get/set methods are turned into property’s to be used as such throughout the program. Properties in Python let you use get/set methods in an object implicitly.

The ‘Instr\_Table’ object holds instances of individual ‘Instruction’ objects. The private variables in the instruction table are: table (list), stack (list), inst\_address. There are 3 get methods for the private variables in the object. Also, an additional 5 methods help manage the instruction table. The ‘gen\_instr’ function generates a new instance of an instruction (initializing the ‘Instruction’ object) object and appends it to the instruction table list. Then, the instruction address is incremented. The pop\_stack method pops and returns an instance off of the object’s stack. The push\_stack method appends an address onto the objects stack list. The back\_patch method pops an instance off of the stack list and assigns it to a temporary variable. This instance is an address stored on the stack. Then, the object’s table is accessed to an index one less than the temporary variable. The operand of the instruction is assigned to a given address. This is used to handle *while* and *if/else* instances in the code. Finally, the print\_table method prints the corresponding assembly code. Accessing the table list within the instruction table object and accessing every entry do this. Every entry in the list is formatted and printed. If the entry’s operand is negative, then only the address and operation is printed. If the entry’s operand is positive, then the address, operation, and operand are printed.

After the objects are defined, the global instances of the symbol table and instruction table are defined. These will be called throughout the functions that manage the syntax analysis. Functions for assignment, expression, expressionPrime, term, termPrime, factory (done actually through the primary function), while, condition, if, compound, read, and write were changed to accommodate the generation of object code. There were no changes made to the syntax analyzer, instead using the instruction table and symbol table methods to generate instructions generated the object code.

**Any Limitation**

None

**Any shortcomings**

There is no check for type matching. If you want to add a Boolean and an int type, this is possible. Boolean can only be assigned as 1 or 0 however if you add an int to the Boolean there is no error.