Second Pass: Creating the Executable

```
REM PROGRAM TO PRINT THE LARGER OF
REM TWO INPUT NUMBERS
REM READ THE TWO INPUT VALUES
IN  FIRST
IN  SECOND
REM DECIDE WHICH IS LARGER BY COMPUTING
REM THE DIFFERENCE
LD  SECOND
SUB  FIRST
REM IF THE DIFFERENCE IS >= 0
REM BRANCH TO PRINT SECOND
BGTR  ELSE
BZ  ELSE
REM THE FIRST IS LARGER
OUT  FIRST
B  FINISH
REM ELSE THE SECOND IS LARGER
ELSE:  OUT  SECOND
FINISH:  STOP
FIRST: DC 0
SECOND: DC 0
```

Program.asm

Location of the instruction in memory

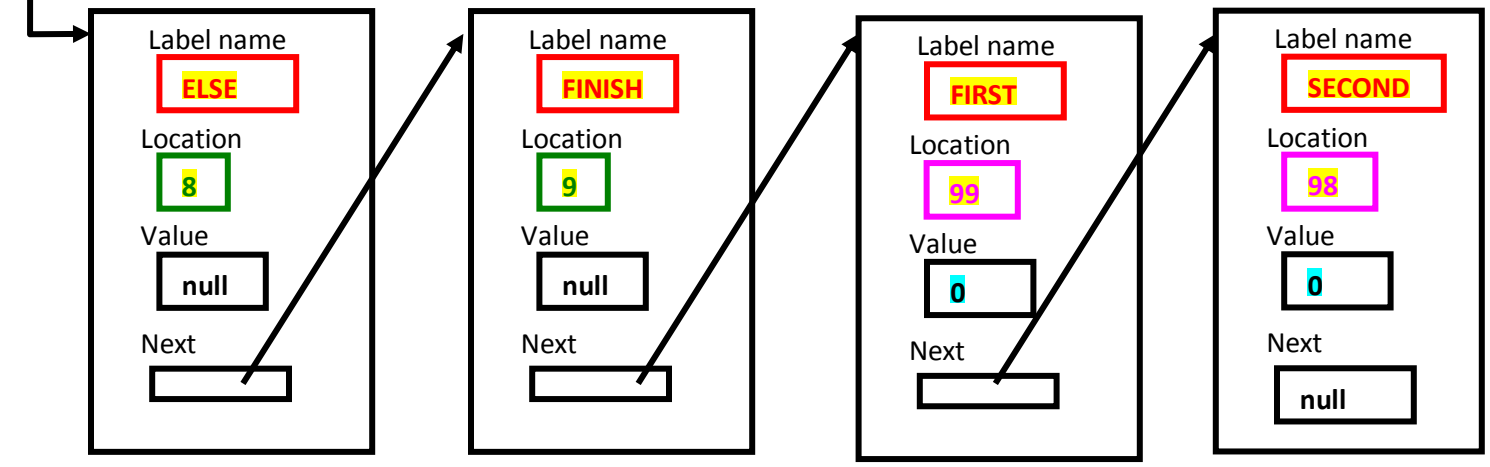
```
Nothing written                0
Nothing written                1
Nothing written
0799
0798                           2
Nothing written                3
Nothing written
0198                           4
0499                           5
Nothing written
1008                           6
1108                           7
Nothing written
0899                           8
0909                           9
Nothing written                10
0898                           11
0000                           12
0                              :
0                              :
0                              94
0                              95
0                              96
0                              97
0                              98
0                              99
```

Data

97

Program.exe

list



Guideline to Compiler Second Pass

Remember that the algorithm for the entire compilation process suggests that you implement 2 methods, *firstPass* and *secondPass*, instead of writing one large method.

Algorithm

First pass: Build symbol table using program file -> return the symbol list

Second pass: Using symbol list and program file, write instructions in executable file

The code for compile should now become:

```
SymbolTable symTbl = firstPass(filename);
secondPass (symTbl, filename);
```

The algorithm for the second pass must use both the symbol table and the program file to write the executable instruction in a binary file. We will assume that the binary file, or executable file, uses the same file name but changes the extension to .ex instead of .asm. For simplicity, we will assume that the program is well written (no syntax errors in the code).

Algorithm

1. Open ASM file for reading (This is a text file).
2. Open/create executable file for writing (This is a binary file).
3. Read a line from ASM file.
    - Is the line a comment or blank?
        - Yes
            - Skip this line and go to step 3.
        - No
            - Continue with this algorithm.
4. Extract first word from the line.
    - Is the first word a label?
        - Yes
            - Is the word a variable?
                - Yes
                    - Decrement the data counter.
                - No
                    - Find the opcode that corresponds to the second word.
                    - Determine the operand (symbol location) from the symbol table using the third word as the label name.
                    - Combine the opcode and operand to make an instruction.
                    - Write the instruction to the executable file.
        - No
            - Find the opcode that corresponds to the first word.
            - Determine the operand (symbol location value) from the symbol table using the second word as the label name.
            - Combine the opcde and operand to make an instruction.
            - Write the instruction to the executable file.
5. Increment the instruction counter.
6. Create a loop that fills the executable file with zeros and increments the instruction counter on each iteration. The loop should stop when the instruction counter becomes equal in value to the data counter.
7. Place the symbols which contain a location value into their respective locations at the end of the executable file.
8. Close the ASM file.
9. Close the executable file.

As previously mentioned, the second pass method will be more readable if some support methods are provided. The Computable interface forces you to create some basic helper methods, however, you are free to create additional methods and/or classes to aid in the completion of this assignment. Additional

helper methods which may prove useful include: reading/writing to files and creating an instruction from an opcode and an operand.

Expectations

Files are a little more complex to manage in the second pass due to the fact that there are two files with distinct objectives: One on which to read, the other on which to write. On the other hand, the extraction of information from the program file (text) is very similar to what is achieved in the first pass.

Again, testing helper methods prior to writing the second pass is critical. In addition, using small steps when programming the second pass will prevent you from straying too far. For example, start by ensuring you can create an executable file by changing the extension only. Once this works, ensure you can read one file at the same time you write the line number in the other one. Once that works, you can modify what you write to include the instructions based on what you read.

Remember to ask for assistance if you have difficulty understanding any component of the assignment. Be absolutely sure you know how to write a list and add/remove items to it for the final exam.

THE USE OF THE DEBUGGER IS HIGHLY RECOMMENDED!