Implemented a simple compiler called COMP that generates A86 code for a sequence of expressions and assignment statements that involve “+”,” *” and “power” operations.

Infix Expression To Post Fix Expression (changer method)

First of all we needed to create a function that turns an infix expression to postfix form (since it will be easier to handle operations in computer in this way). Changer function does that for us, as follows:

Takes its parameter “input” from the current object that method is running in. Does necessary error handling operations at the beginning of the method, then takes the whole expression that we want to change to the Postfix form and traverses the chars in that string one by one. Whenever it faces a variable, it pushes it to the output string (a “.” sign comes before all variables and “,” sign cmes before all operator signs, in order to easen some further operations for us.) As it faces an operator sign as:

- “) ” : prints all operators in stack to the output by popping, untill seeing “(“ in the stack. The reason is to give precedence to “( )“.
- “ (“ : just pushes “ (“ to the stack

Opr method:

“ * ” “ + ” : untill seeing “(“ in the stack, compares the current character’s and the last stack item’s precedence values (which are defined before).

→If stack’s item has higher precedence prints it to output. Then pushes current sign to the stack and ends the method.

→If current sign has higher precedence than stack’s last item, pushes stack’s item back, then pushes our current sign to the stack then ends the method.

End of Opr method.

- “ , ” : acts like we see “) ” “ ^ ” “ (“ consecutively. For “^”, applies the Opr method, with higher precedence for ^ sign than any other signs (the reason will be explained in “p” part.)
- “ p ” : whenever it faces with “p” just moves iterator to check inside of the pow() statement, so as to understand whether its a power method or not we look for “pow(“ starting. If we are sure that is a power, we are changing pow(a,b) statement into the form ((a)^(b)). We use already existed paranthesis of method definition as normal paranthesis, and when we see “ , “ we change it to “)^(“. We handle outer paranthesis’ task by giving “ ^ “ symbol higher precedence than others.
- “Facing with other characters”: If the char we look at is different from any characters above, we check whether its a proper variable name or not. If it is, we just add it to the output string after a “.” Sign, which indicates a variable coming after.

End of changer method

As we read the file, in given format, if there is an equal sign we handle variables accordingly. if the current variable is in the left hand of the equal sign side, we do necessary (defined) operations given in the right hand side part, and pushing that variable's value accordingly then necessary codeparts for Assembly are be pushed into a huge "DecVar" string. When we face with a variable, we check for its' existance via a map that we created. If they were not created before, we initialized them as "0". So as to operate all operations in Assembly, whichs order are defined by the postfix expression, necessary codeparts for Assembly are be pushed into a huge "DecCalc" string. Whenever there is no equal sign which means the given expression is expected to be printed, we calculate the given expression and assign the calculation to a new variable then we add necessary "print" order to DecCalc string. At the end of the java project, they will be gathered and create the file that will be executed by the assembler.

The Assembly Part:

In the Assembly, A86 part of the code; we needed to do power operations, initializing, adding, multiplying and printing and at the end, exiting the program.

We divided each variable as having high and low parts, and we pushed values accordingly, last 16 bits as low, rest as high (max 32 bits). We created all variables in var+" variable no" +_H and var+" variable no" +_L format

Addition:

We added low part to low part, then we used "add with carry" (adc) when adding high parts. Then we pushed first high part of the result then the low part.

Multiplication:

We first multiplicated low1 to low2, carry of this stored in dx, then we multiplicated high1 low2 then we added carry to this, then multiplicated high2 low1 and added this result to last addition. Since high1*high2 is bigger than 32 bits, we did not take it into consideration. Then we pushed first high part of the result then the low part.

Pow. Operations:

We consequently did multiplication operations. We handled the case of x^0 as being "0". We did not use high part as power as it exceeds 32 bits. As we are jumping during multiplication operations, we used a counter which shows current power operations. Then we pushed first high part of the result then the low part.

Printing:

We used print counter for the same reason as the power operation. We used the algorithm that we learnt in class, for Hexadecimal printing, which is also an old exam question. As we know each hexadecimal number consists of 4 bits so we have loop which operates 4 times. Then we rotate left the 16 bits number just 4 bits to print the left most 4 bits which is also left most digit of the hexadecimal number. Before printing the digit, method checks whether the to be printed digit is a number or a alphabetic character due to hexadecimal numbers consist "a,b,c,d,e,f".

End of the Assembly part

Executing on Linux

First of all we created the necessary executable java.class file with the command *“javac Organiser.java*” which creates executable *Organiser.class* file.

Next command line is in the “*java Organiser input.txt output.asm*” format, which executes the Organiser and creates the output file.

As we have the output file in .asm format, we use the A86 compiler to execute this output file, which in the end prints the console expected output values. The command lines we used consecutively are:

A86 output.asm

output

(Since we have the A86 compiler in DosBox, we used DosBox command lines in order to reach the result.)