CMPE 230 ASSIGNMENT 1 REPORT

Elif Çalışkan 2016400183

Yağmur Kahyaoğlu 2015400057

HOW TO COMPILE:

In a terminal, call commands:

>cmake CMakeLists.txt

>make

Then you can run the code with the following command:

>./project1 [input.co] [output.asm] [temp.txt]

1)PROBLEM DESCRIPTION

We are asked to implement a compiler called COMP that generates A86 code for a sequence of expressions and assignment statements that involve +, * and power operations. We are given a list of expressions in infix form. There may be some errors in given expression, we should also find the error and print an error statement with the relevant number of line. First we need to convert the infix expression into postfix order. While converting, we should check if there is an error. If there is an error, the program should terminate and print an error statement. If there is no error, then we should move into a method which writes an assembly code. Here we can call methods of addition, multiplication, power. After every line we should assign the value to the given variable. At the end of execution we should print the values of wanted variables and terminate the asm program. Since we will be given 32 bit constants and the results will also be 32 bit, we should split every 32 bit number into two sections to keep them in registers. For example lets say x1=1abcdh. x1high will be 0001, x1low will be 0abcdh. We will be calling every constant by using this notation.

2)PROBLEM SOLUTION

We used 15 methods to solve this problem.

1)int main(int argc, char* argv[]): This is the main method. It takes 4 argv values. First one is the name of program, second one is the path of input file. Third one is the path of output file and the fourth one is a file we use to store a part of the output temporarily during the process. In main method, we keep an integer numLine which is the counter of lines. We create a vector of strings which keeps the variable names. Then we start reading the input file. We look for the "=" sign. If there is an assignment it means we are not looking for an output value, so we need to evaluate the expression. We split the line into two sections first one is the variable, second one is the expression. We call the updateStr() method to eliminate space

characters and errors. If there is no assignment, it means we need to print the output value. After writing outputhigh and outputlow values, we print one empty line. If there is an assignment, we declare these variables by adding low and high at the end. We also add a's or z's to the end of the variable names to handle the case sensitivity problem. Then we push them to the stack in assembly code. We convert the expression into postfix and if there is no error, we call toAssembly method to compute the variable's value.

2)string updateStr(string& str, int numLine, bool& error): This method updates the given string. It ignores the space characters and looks for any errors. It takes a string, an integer and a bool. str is the expression which will be updated, numLine is the line number, error is the bool which will be true if there is an error. Since we want to know if there is an error, the bool value is passed by referance. We have two integers inside the method: lastOne, now. Now keeps the current value and lastOne keeps the previous value. Every character has different values for now. For example; '*' has value 2, '(' has value 3. By keeping two integers we ignore the blank spaces and we can find the errors. For example; "x+)" or "z(". After the update, it returns the new string.

3)stack<string> infixToPostfix(string expr, vector<string> vals, int numLine): This method is used for converting an infix expression into a postfix expression. It takes a string, a vector and an integer. expr is used for the whole expression in line, vals is the vector of known variables, numLines is the line number. This method converts the stack by calling the expression method and starting a recursive process. After converting the expression into postfix, the method adds a's or z's to the end of the variable names to handle the case sensitivity problem and checks if the given variable is used before; if it is not used, it is replaced with "0". Method returns the stack after updating unknown variables.

4)void expression(string expr, stack<string>& postf, int numLine):This method is called by infixToPostfix method. It takes a string, a stack and an integer as parameters. expr is used for the expression, postf is the main stack which keeps the postfix representation, numLine is the line number. This method splits the expression into two sections: term and moreterms. To split the expression, it looks for '+' sign for determining a moreterm. If there is no '+' sign in expression, it means the whole string is a term, so it calls term method. If there is a '+'sign and we can separate it from the '+' sign without disturbing the equality of parenthesis, it calls term with the first part and moreterms with the second part of the expression.

5)void term(string expr, stack<string>& postf, int numLine): This method is used for infixToPostfix method. It takes a string, a stack and an integer as parameters. expr is used for the expression, postf is the main stack which keeps the postfix representation, numLine is the line number. This method splits the expression into two sections: factor and morefactors. To split the expression, it looks for '*' sign for determining a moreterm. If there is no '*' sign in expression, it means the whole string is a factor, so it calls factor method. If there is a '*' sign and we can separate it from the '*' sign without disturbing the equality of parenthesis, it calls factor with the first part and morefactors with the second part of the expression.

6)void moreterms(string expr, stack<string>& postf, int numLine): This method is used for infixToPostfix method. It takes a string, a stack and an integer as parameters. expr is used for

the expression, postf is the main stack which keeps the postfix representation, numLine is the line number. This method splits the expression into two sections: term and moreterms. To split the expression, it looks for '+' sign for determining a moreterm. If there is no '+' sign in expression, it means the whole string is a term, so it calls term method and pushes a '+' sign into the stack. If there is a '+'sign and we can separate it from the '+' sign without disturbing the equality of parenthesis, it calls term with the first part, pushes '+'sign to stack and moreterms with the second part of the expression.

7)void morefactors(string expr, stack<string>& postf, int numLine): This method is used for infixToPostfix method. It takes a string, a stack and an integer as parameters. expr is used for the expression, postf is the main stack which keeps the postfix representation, numLine is the line number. This method splits the expression into two sections: factor and morefactors. To split the expression, it looks for '*' sign for determining a morefactor. If there is no '*' sign in expression, it means the whole string is a factor, so it calls factor method and pushes a '*' sign into the stack. If there is a '*'sign and we can separate it from the '*' sign without disturbing the equality of parenthesis, it calls factor with the first part, pushes '*'sign to stack and morefactors with the second part of the expression.

8)void factor(string expr, stack<string>& postf, int numLine): This method is used for infixToPostfix method. It takes a string, a stack and an integer as parameters. expr is used for the expression, postf is the main stack which keeps the postfix representation, numLine is the line number. If expr has two parentheses around it, method calls the expression with the expr without parenthesis. Else, it means expr is id, num or pow so it pushes expr to stack.

9) void toAssembly(stack<string>& expr, vector<string>& vals ,int numLine, int m, int& counter): This method is used for converting the postfix expression into assembly commands. It takes a stack, a vector and two integers. expr is used for postfix expression, vals is vector of used variable strings, numLine is the line number and m is a label for understanding if it is called from power or not. If we call from power m's value is 0, else it is 1. We use m value in an if statement and if it is zero we don't call the assign function. In this method, first we pop the first element of the stack. If it has '+' we call addition, if it has '*' we call multiplication. If it has 'pow', it means there are two expressions in pow and we call infixToPostfix for both expressions and call toAssembly code with parameter 0.Then, we call power. Else it means we have either a variable or a number. So we look for that string in vals vector. If the string is in vals, we push the high and low values into stack in assembly code, else we put '0's at the beginning of the number and push the high and low values into stack in assembly code. At the end if m equals to 1, it means toAssembly is called from a line, so assign method should be called.

10) void addition(int counter): This method is used for addition in assembly code. It takes one integer parameter. numLine is used for changing the label names and eliminating the duplicates. It adds the two low values first and adds the carry to the addition of high values. Low and high values are pushed into the stack in the assembly code.

11) void multiplication(int numLine): This method is used for multiplication in assembly code. It takes one integer parameter. numLine is used for changing the label names and

eliminating the duplicates. It multiplies the two low values first. It multiplies the first low value with the second high value and the first high value with the second low value. Then it adds the carry of the first multiplication to the high values. Low and high values are pushed into the stack in the assembly code.

12) void power(int& counter): This method is used to take the given power of an expression. It checks if the power or the base part is 0 or 1 and pushes the right values accordingly, else it multiplies the base part power times with a loop using the same algorithm used in the multiplication method. Then, low and high values are pushed into the stack in the assembly code. It takes one integer parameter. numLine is used for changing the label names and eliminating the duplicates.

13) void assign(): This method makes the assignment operation. In this method, the addresses of high and low values are assigned in bx register. It takes one integer parameter. numLine is used for changing the label names and eliminating the duplicates.

14) void print(string val, int numLine,int a): This method is used for printing the value of the output in assembly code. It takes one string and two integers. val is the name of the wanted variable, numLine is the line number, a is used for putting different names for high and low labels. We got some help from the previous exam.

15)void printError(int numLine): This method is used for printing an error message in assembly code. It takes an integer parameter which is the line number. Inside the method, it concatenates the numLine into the error message.

3)INPUT

Exampe 1:

```
x = 3abd
y = (x+0e)*4
pow(y,1)
x
```

Example 2:

```
k = 5*  (7b+1)
l = ((8 + x)*pow(3,0)
k
l
```

4)OUTPUT

Example 1:

```
DATA SEGMENT
xalow  dw 0
xahigh dw 0
yalow  dw 0
yahigh dw 0
```

```
alph6low   dw 0
alph6high  dw 0
DATA ENDS
PUSH offset xahigh
PUSH offset xalow
PUSH 00000h
PUSH 03abdh
POP AX
POP CX
POP BX
MOV [BX],AX
POP BX
MOV [BX],CX
PUSH offset yahigh
PUSH offset yalow
PUSH xahigh
PUSH xalow
PUSH 00000h
PUSH 0000eh
POP CX
POP DX
POP AX
POP BX
ADD AX,CX
JC CARRY2
DONECARRY2:
ADD BX,DX
PUSH BX
PUSH AX
JMP CARRYON2
CARRY2:
ADD BX,1
JMP DONECARRY2

CARRYON2:
PUSH 00000h
PUSH 00004h
POP AX
POP BX
POP DX
POP CX
PUSH BX
PUSH DX
PUSH AX
PUSH CX
MUL DX
PUSH DX
PUSH AX
POP BX
POP CX
POP AX
POP DX
MUL DX
ADD CX,AX
POP AX
POP DX
MUL DX
ADD CX,AX
PUSH CX
PUSH BX
POP AX
POP CX
POP BX
MOV [BX],AX
POP BX
MOV [BX],CX
PUSH offset alph6high
PUSH offset alph6low
```

```
PUSH yahigh
PUSH yalow
PUSH 00000h
PUSH 00001h
POP CX
POP DX
POP AX
POP BX
CMP DX,0
JZ MIDDLE111
JMP MIDDLE211
MIDDLE111:
CMP CX,0
JZ OUT211
CMP CX,1
JZ OUT311
JMP MIDDLE211
MIDDLE211:
CMP AX,0
JZ OUT111
CMP AX,1
JZ OUT211
SUB CX,1
PUSH BX
PUSH AX
FOR11:
PUSH BX
PUSH AX
POP AX
POP BX
POP BX
POP DX
PUSH AX
MUL DX
MOV DX,BX
POP BX
PUSH BX
PUSH AX
MOV AX,BX
MUL DX
POP BX
ADD DX,BX
ADD DX,BX
POP BX
PUSH DX
PUSH AX
MOV AX,BX
MOV BX,0
CONT11:
LOOP FOR11
JMP DEVAM11
OUT111:
PUSH 0
PUSH 0
JMP DEVAM11
OUT211:
PUSH 0
PUSH 1
JMP DEVAM11
OUT311:
PUSH BX
PUSH AX
JMP DEVAM11
DEVAM11:
POP AX
POP CX
POP BX
MOV [BX],AX
```

```
POP BX
MOV [BX],CX
PUSH alph6high
POP BX
MOV CX,4h
MOV AH,2h
LOOP131:
MOV DX,0fh
ROL BX,4h
AND DX,BX
CMP DL,0ah
JAE HEXDIGIT31
ADD DL,'0'
JMP OUTPUT31
HEXDIGIT31:
ADD DL,'A'
SUB DL,0ah
OUTPUT31:
INT 21h
DEC CX
JNZ LOOP131
ENDLOOP231:
MOV AX,0

PUSH alph6low
POP BX
MOV CX,4h
MOV AH,2h
LOOP132:
MOV DX,0fh
ROL BX,4h
AND DX,BX
CMP DL,0ah
JAE HEXDIGIT32
ADD DL,'0'
JMP OUTPUT32
HEXDIGIT32:
ADD DL,'A'
SUB DL,0ah
OUTPUT32:
INT 21h
DEC CX
JNZ LOOP132
ENDLOOP232:
MOV AX,0

MOV DL,10
MOV AH,02h
int 21h
PUSH xahigh
POP BX
MOV CX,4h
MOV AH,2h
LOOP141:
MOV DX,0fh
ROL BX,4h
AND DX,BX
CMP DL,0ah
JAE HEXDIGIT41
ADD DL,'0'
JMP OUTPUT41
HEXDIGIT41:
ADD DL,'A'
SUB DL,0ah
OUTPUT41:
INT 21h
DEC CX
JNZ LOOP141
```

```
ENDLOOP241:
MOV AX,0

PUSH xalow
POP BX
MOV CX,4h
MOV AH,2h
LOOP142:
MOV DX,0fh
ROL BX,4h
AND DX,BX
CMP DL,0ah
JAE HEXDIGIT42
ADD DL,'0'
JMP OUTPUT42
HEXDIGIT42:
ADD DL,'A'
SUB DL,0ah
OUTPUT42:
INT 21h
DEC CX
JNZ LOOP142
ENDLOOP242:
MOV AX,0

MOV DL,10
MOV AH,02h
int 21h
JMP exit
exit:
MOV AH,4ch
MOV AL,00
int 21h
```

Result:

0000EB2C

00003ABD

Example 2:

```
code segment
MOV BX,MSG
MOV CX,20d
MOV AH,02h
MORE:
MOV DL,[BX]
int 21h
inc BX
dec CX
JNZ MORE
int 20h
MSG:
DB 'Line 2: Syntax error'
code ends
```

Result:

Line 2: Syntax error

5)CONCLUSION

To sum up, we created a cpp project which implements a compiler called COMP that generates A86 code for a sequence of expressions and assignment statements that involve +,* and power operations. First we converted the infix ordered expression into postfix order. Then we wrote an assembly code to evaluate the expression. While writing the assembly code we used some methods for addition, multiplication, power and assignment operations. After writing the asm file, we ran it on DOSBox. While working on this assignment we had many errors but we learned how registers work and how to implement a code in assembly. We had to look from many perspectives to find solutions to our problems. But our code worked in our testcases and found the right answers.