CmpE 230

Spring 2018 – Homework 1

Creating a compiler that transforms from COMP to Assembly

Cemal Aytekin – 2015400126

Berke Esmer – 2015400021

1. Description

In this project, we implement a simple compiler called COMP that generates A86 code for a sequence of expressions and assignment statements that involve +,* and power operations.

If the input has an error syntactically, the program will print an error message on a console such that “Syntax Error : line number”. Otherwise it will compute the result and print what is asked.

2. Project Implementation

We implemented the code in 4 stage. Those are,

- ✔ Reading input & Error checking
- ✔ Transforming infix into postfix notation
- ✔ Calculating the arithmetic operations
- ✔ Printing the result

2.1. Reading input & Error checking

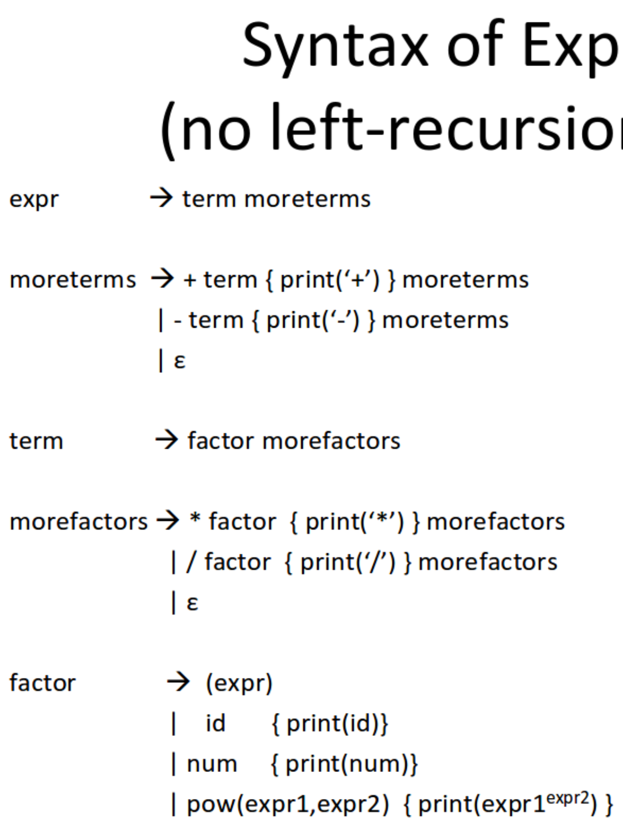
We took the input file as argument in the terminal. Then we started a while loop that iterates until the end, reading every line.

If any line contains an error such as unbalanced parantheses, it gives an syntax error highlighting the line number that has the error. Otherwise, it initializes the variables at the first part of assembly code and goes to next stage.

The important case here is to split the number into two variables to be able to use 32bit number system. (E.g. VARIABLE_HIGH, VARIABLE_LOW)

2.2. Transforming infix into postfix notation

In this stage, we take the expression line as the parameter of functions. Then we determine the left and right hand sides. After all, we transform the given infix expression into postfix notation by using the below BNF notation.



Syntax of Expression
(no left-recursion use this)

expr → term moreterms

moreterms → + term { print('+') } moreterms
| - term { print('-') } moreterms
| ε

term → factor morefactors

morefactors → * factor { print('*') } morefactors
| / factor { print('/') } morefactors
| ε

factor → (expr)
| id { print(id)}
| num { print(num)}
| pow(expr1,expr2) { print($expr1^{expr2}$) }

Picture 1: BNF notation of postfix transformation

2.3. Calculating the arithmetic operations

After constructing the postfix notation, we create a queue by pushing every element. Then in a while loop, we pop them and process them.

Based on the popped element, there are three options

- The popped element is a hexadecimal number
- The popped element is a variable
- The popped element is an operator (+, *, pow)

In the first case, the necessary Assembly code is written onto the output file to push the number into Stack.

In the second case, the variable adress is pushed into Stack.

In the final case, the necessary Assembly code pops the last pushed two numbers and makes the arithmetic operation based on the operator.

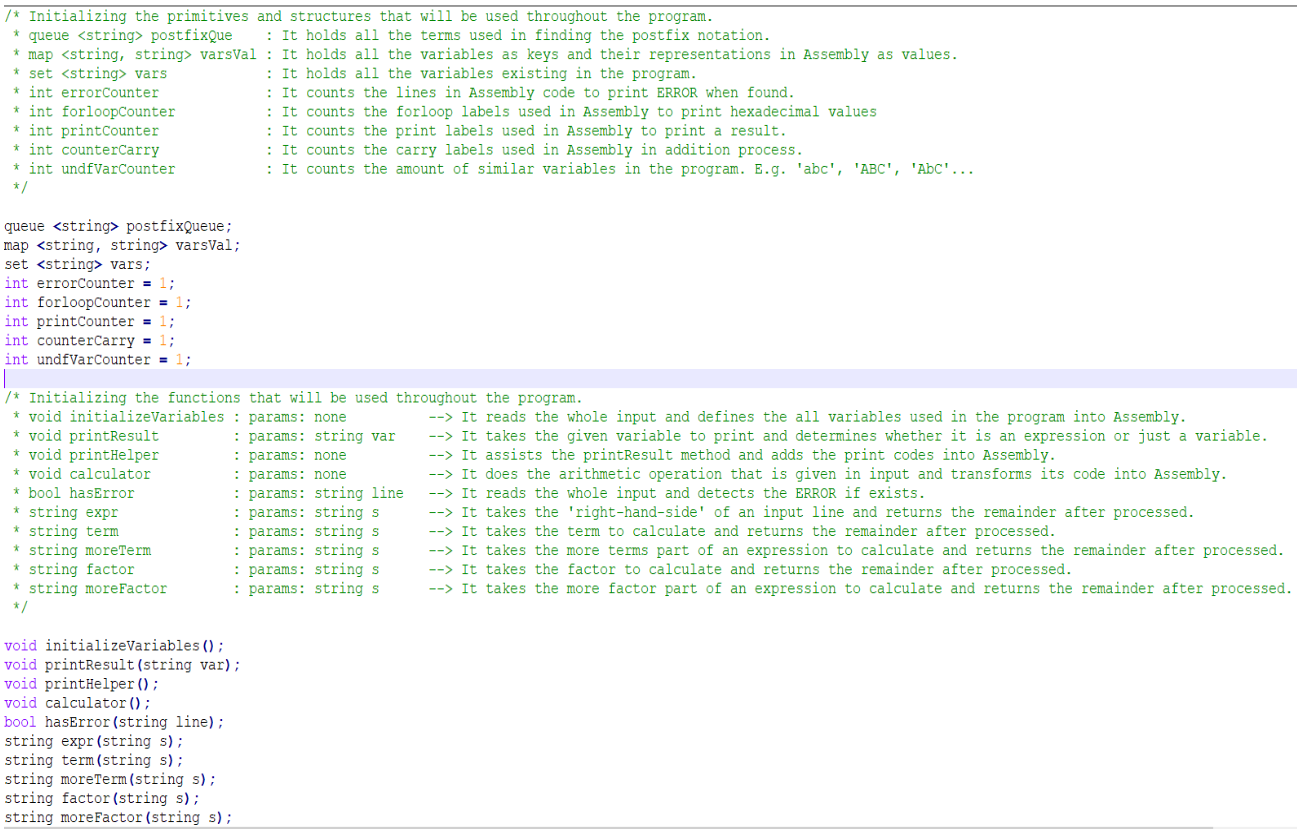
2.4. Printing the result

As the input file comes to the printing, we determine whether it is an expression or just a variable name.

If the print action is called for an expression, we evaluate the expression as stated above, otherwise we just call the value of variable in Assembly, then we print it.

The output will have 8 digits representing the 32bit hexadecimal number.

3. Functions & Global Variables

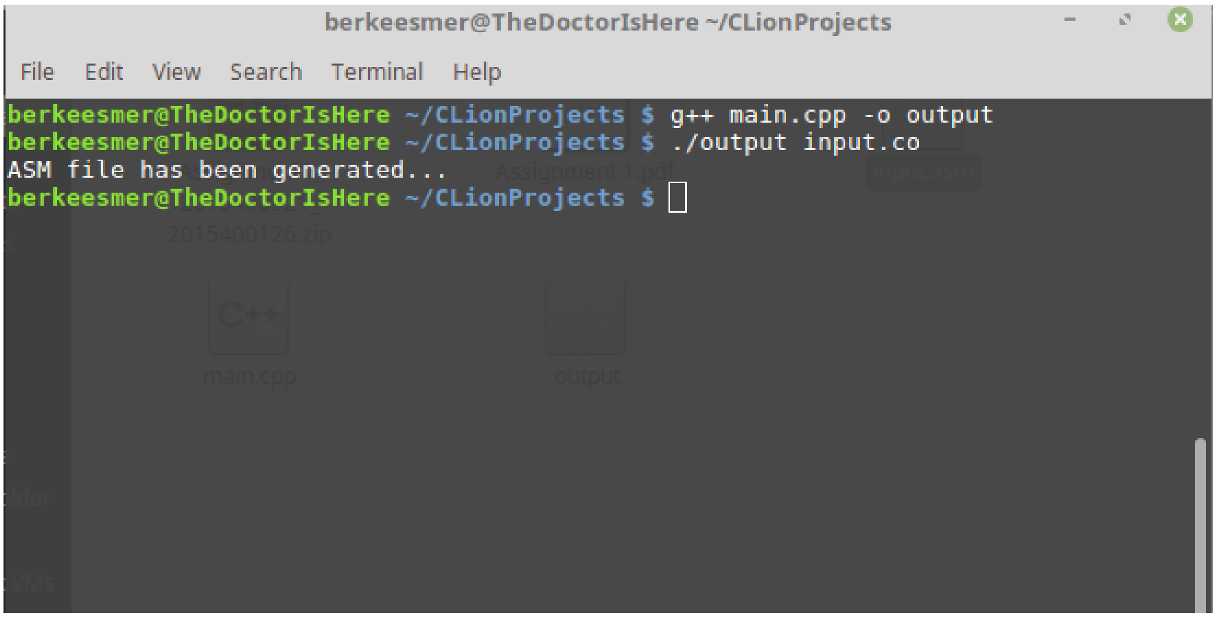


```
/* Initializing the primitives and structures that will be used throughout the program.
 * queue <string> postfixQue    : It holds all the terms used in finding the postfix notation.
 * map <string, string> varsVal : It holds all the variables as keys and their representations in Assembly as values.
 * set <string> vars            : It holds all the variables existing in the program.
 * int errorCounter             : It counts the lines in Assembly code to print ERROR when found.
 * int forloopCounter           : It counts the forloop labels used in Assembly to print hexadecimal values
 * int printCounter             : It counts the print labels used in Assembly to print a result.
 * int counterCarry             : It counts the carry labels used in Assembly in addition process.
 * int undfVarCounter           : It counts the amount of similar variables in the program. E.g. 'abc', 'ABC', 'AbC'...
 */

queue <string> postfixQueue;
map <string, string> varsVal;
set <string> vars;
int errorCounter = 1;
int forloopCounter = 1;
int printCounter = 1;
int counterCarry = 1;
int undfVarCounter = 1;

/* Initializing the functions that will be used throughout the program.
 * void initializeVariables : params: none         --> It reads the whole input and defines the all variables used in the program into Assembly.
 * void printResult         : params: string var   --> It takes the given variable to print and determines whether it is an expression or just a variable.
 * void printHelper         : params: none         --> It assists the printResult method and adds the print codes into Assembly.
 * void calculator          : params: none         --> It does the arithmetic operation that is given in input and transforms its code into Assembly.
 * bool hasError            : params: string line  --> It reads the whole input and detects the ERROR if exists.
 * string expr              : params: string s     --> It takes the 'right-hand-side' of an input line and returns the remainder after processed.
 * string term              : params: string s     --> It takes the term to calculate and returns the remainder after processed.
 * string moreTerm          : params: string s     --> It takes the more terms part of an expression to calculate and returns the remainder after processed.
 * string factor            : params: string s     --> It takes the factor to calculate and returns the remainder after processed.
 * string moreFactor        : params: string s     --> It takes the more factor part of an expression to calculate and returns the remainder after processed.
 */

void initializeVariables();
void printResult(string var);
void printHelper();
void calculator();
bool hasError(string line);
string expr(string s);
string term(string s);
string moreTerm(string s);
string factor(string s);
string moreFactor(string s);
```

Picture 2: All functions and global variables

4. How to Compile

In the terminal, the following commands will first create an executable comp file, then based on the given input file path, it will compile and create the output file containing Assembly code.


berkeesmer@TheDoctorIsHere ~/CLionProjects
File Edit View Search Terminal Help

```
berkeesmer@TheDoctorIsHere ~/CLionProjects $ g++ main.cpp -o output
berkeesmer@TheDoctorIsHere ~/CLionProjects $ ./output input.co
ASM file has been generated...
berkeesmer@TheDoctorIsHere ~/CLionProjects $ 
```

Picture 3: Necessary commands to compile and run the program

$ g++ main.cpp -o output

$ ./output input.co

5. Conclusion

The target was to convert the given set of expressions into Assembly code that can handle the arithmetic operations.

It was a great project to understand how Assembly works and how it is coded. So it was really helpful.

Also, studying the project in a 2 people-team was a great experience. We believe that, some projects can be understood better if they are assigned as the teamwork.