Implementation of an expression compiler

Khanjan Desai (MT2016506)

Mandeep Singh (MT2016513)

Prajwal Sharma K (MT2016519)

Sathyam Panda (MT2016525)

December, 2017

Introduction

In this project we implement a parser. This parser reads in an Infix expression and converts it to postfix expression to evaluate it. The Infix expression is given in C and in the process of evaluation we generate an ARM assembly language code which is compatible with KEIL tool.

Implementation

C code is written to take Infix expression as input and compute Postfix expression from the given input. This program gives ARM assembly language instructions as an output. The C program can be executed in following ways from command line -

1. `./compile <expression> <filename.s>` - This generates ARM assembly code in a file by name *filename.s* for the given expression. Here, expression to be evaluated is also passed through command line.
2. `./compile <filename.s> -` The ARM assembly code is generated in the file *filename.s* for default expression given in the code.
3. `./compile –` When this command is executed the default expression is used to create ARM assembly language instruction in the default file by name *asm_code.s*

The C code implements logic for Infix to Postfix conversion with the help of Stack data structure. The expression is sequence of characters. Now, we need to convert it to an array of operands and operators. So, we created a structure that takes value and type. The type can be operand or operator. Once, we get this array of value then, this can be used for Infix to Postfix conversion. After the expression is in Postfix form analyzing it produces an ARM assembly language code into a desired file.

The steps followed is implementing the project is shown in Figure 1.

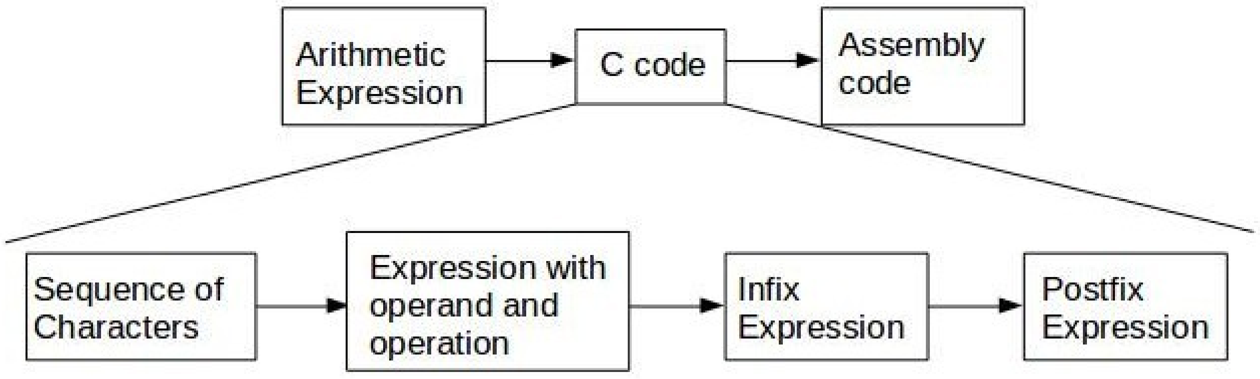

Fig. 1. Steps in Implementing an arithmetic expression parser.

Challenges Faced

Floating point values were a challenge to detect from expression given as sequence of characters. Because, floating point value had '.' which by itself was an alphanumeric character. We faced problem in detecting negative numbers as well. For example, 5 * - 3 expression could not be recognized initially. So, during conversion from sequence of characters to array of values expression was changed to, 5 * (0 – 3). Hence, both the problems were solved.

Conclusion and Future Scope

An compiler that parses arithmetic expression was built with features of floating point operation and divide by zero detection long with ability to evaluate expression of any length. We assume that the user inputs the expression with equal number of open and close brackets. If the user does not, then an error will be generated. Parsing power and modulo operation features are yet to be added into this compiler.